Amiga **LOGO**

Tutorial
and Reference

Amiga Logo software and manual **developed** and written **by Carl Sassenrath.**

Amiga Logo is dedicated in memory of Elizabeth R. Smyth.

Many thanks to Cynthia L. Sassenrath and Robert J. Mical, without whom this product could not have been created.

Contents

Here's how to test drive Amiga Logo:

- Check that your Amiga Logo disk is write protected. *The little tab in the corner of the disk should be in the open position.*
- Insert your Amiga Logo disk into disk drive 0.
- Turn on or restart your Amiga.
- Double click on the Amiga Logo disk icon. This will open the main disk directory.

AmigaLogo

- Double click on the Logo drawer icon. This will open the Logo drawer.

Logo

- Double click on the DEMO icon.

DEMO

You will now see Amiga Logo come to life.

To stop the demo, move the mouse to the top border of the screen, click the right mouse button, and select QUIT from the PROJECT menu.

If you want to try your own hand at Logo, click on the AmigaLogo icon.

AmigaLogo

Logo is a computer language originally designed to introduce young minds to the world of computers. Amiga Logo hosts an interactive, discovery-oriented environment, which makes Logo fun to learn and easy to use. Like other modern computer languages, Logo offers features such as built-in graphics and list processing.

Although learning to program with Logo is fun, computer programming is an engineering discipline that requires knowledge of the language's principles, attention to details, and plenty of practice. Computation belongs to the realm of applied mathematics, and learning to program computers is a demanding discipline. Remember that *thought should always precede experimentation.* The computer is only a machine; with practice and patience, you will learn to master Logo and make the computer do what you want.

Many dialects of Logo exist. Amiga Logo was designed to be very similar to the original Logo. It provides the same basic vocabulary and supports the fundamental features and primitives of this original Logo. The widely available Logo textbooks found in most bookstores will work well for learning and teaching the language. If you are familiar with other Logos, you will find that Amiga Logo has extended features to make use of the superior graphics capabilities of the Amiga.

Amiga Logo runs on all Commodore Amiga computers. One disk drive and 512K bytes of memory are required for basic operation; however, with more memory Logo will perform faster and display more colors. We highly recommend the use of a color monitor.

Some of these files will not be seen from the Workbench, as they do not have icons. You will need to use the CLI to copy or move these files.

To copy Amiga Logo to a hard drive, click on the hard drive disk icon to open the hard drive window. Point your mouse cursor to the Logo drawer icon, click, and drag the icon from the Amiga Logo Disk window to the hard drive window. Release the mouse button to copy the drawer to the hard drive.

Another way to copy files to the hard drive is to use the CLI. For example, to copy the contents of the Logo drawer from the Amiga Logo disk to the hard drive, you would type the following:

Copy AmigaLogo:Logo to dH0:directoryname all

You can make or specify a destination directory on the hard drive to which the file will be copied. Refer to your AmigaDos or Hard Drive manuals for further information about the CLI. Repeat the above procedure for every file you want to copy. You must copy the MathTrans.Library file for Amiga Logo to work. This file is found in the LIBS directory of the Amiga Logo system disk.

What's Included on the Amiga Logo Disk

The following is a description of all the icons you will find on your Amiga Logo System disk.

System Drawer

CLI - updated to Shell	Command Line Interface for entering AmigaDos commands
DiskCopy	Copy utility
Format	Format a floppy disk
FastMemFirst	Allows programs to use fast memory before $C00000 memory, resulting in faster system operation.
SetMap	Allows you to select the correct keymap for your keyboard.

Amiga Logo comes ready to run on a standard 1.3 Workbench disk. Amiga Logo is distributed with Workbench 1.3, but it will also run under version 1.2. However, the speech feature of Amiga Logo requires 1.3. You can run Logo from this disk directly, but to avoid accidental damage, we suggest that you make a copy first. See your *Introduction to the Commodore Amiga* user's manual for instructions on duplicating a disk. Once you've made a copy, store the original in a safe place away from heat and moisture.

If you want to copy Logo to another floppy or a hard disk, you will need to know what files to copy. Below is a list of the relevant files that come with your Logo:

AmigaLogo   is the Logo main program. This file is required to run Logo.

INIT   is an initialization file used by Logo when it starts running. It is not absolutely required in order to run Logo, but it contains a number of helpful commands. You can add your own commands to this file to customize your Logo.

DEMO   is the Logo demo file. It is not required for the proper operation of Logo, but presents an interesting demonstration of Amiga Logo. You may want to load this file and look at it. It contains a number of Logo examples.

MathTrans.Library is a library file found in the LIBS directory of the system disk, and it comes on the Workbench disk with every Amiga system. This file **must** be present for Logo to work.

InitPrinter	Initializes your printer using the printer settings specified in Preferences.	

Empty Drawer

An empty directory that can be copied and renamed to create new drawers.

Prefs Drawer

Preferences	The main preferences window
CopyPrefs	Copies the devs:system-configuration to df0:devs
Pointer	Edit pointer window
Printer	Change printer window
Serial	Change serial window

Logo Drawer

DEMO	An Amiga Logo demonstration program. Double click to run this program.
INTRO	An audio-visual introduction to Logo programming. Double click to run the program.
INIT	Initialization file used by Logo when it starts running.
AmigaLogo	This is the Logo main program. Double click on the icon to run Logo.

Install printer   *This allows you to add the printer driver you need from Exti*

About this Manual

This manual serves dual purposes. First, it gives the Logo newcomer a taste of the programming language and its environment. Second, it serves as the official reference manual for day to day programming. This manual assumes that you are already familiar with the basics of using the Amiga. This includes the use of the Workbench, disk, mouse, and keyboard. If you are unfamiliar with any of these components, refer to your Amiga DOS User Manual and your Amiga Installation Manual.

If you are a beginner, learning a new language is no modest undertaking. Whether it be a human or computer language, you've got a task ahead of you. There are many books about learning Logo. You can find them in almost any bookstore or public library. This manual is meant to supplement other sources of instruction.

This user's manual is divided into two major parts: a tutorial and a reference.

The tutorial sections give you a brief introduction to the Logo language. They describe the basic elements of the language and the operation of the Logo programming environment. In addition to the examples in the tutorial it is recommended that the novice Logo programmer obtain another tutorial type text on the language. Since Amiga Logo is similar to the original Logo, a number of tutorial books are widely available and the examples should work with Amiga Logo.

If you are already familiar with Logo, it is suggested that you use the tutorial for a quick review. After you have warmed up, use the reference section for detailed descriptions of all the Logo functions.

Using Logo

This tutorial section gives a brief introduction to using Amiga Logo. It gets you started by showing you how to run Logo, enter and edit command lines, switch between the graphics and text windows, and exit back to DOS.

Once you have learned these steps, the next section will teach you the language.

Running Logo

Start by making a backup of your Amiga Logo distribution disk. You wouldn't want to accidentally damage it. To make a backup, use the Workbench or CLI diskcopy operations. See your *Introduction to the Commodore Amiga* user's manual for more information.

Once you've made a copy, store the original in a safe place away from heat and moisture.

To start Logo from the Workbench, double click on the AmigaLogo program icon. After a few seconds, the screen will turn black and a Logo window will appear.

Double click here to run AmigaLogo ————

AmigaLogo

To start Logo from the CLI, type AmigaLogo:LOGO/ AMIGALOGO. If this is followed by a Logo file name, the file will be loaded as soon as Logo starts running.

When Logo starts running it will present you with two windows, one of which is titled AMIGA LOGO - COMMAND. This is a textual *Command Window* in which you will type your commands and create new Logo programs.

```
┌─AMIGA LOGO──────────────────────────────────────┐□□│
```

```
┌─┐ AMIGA LOGO - COMMAND ═══════════════════════
Amiga Logo, Version 1.00
Copyright (C) 1989 Commodore Amiga, Inc.
Copyright (C) 1989 Carl Sassenrath
All rights reserved.

? █
```

This window is similar to the other windows that you have used on the Amiga. You can change its size and position to suit your fancy. You can also make the window disappear in order to display the *graphics* window beneath. We will discuss the ways to control windows later.

The command window is also called the *Text Screen* in this (and most other) Logo books. Do not confuse the Logo screens with what the Amiga calls a screen. Logo screens are really just Amiga windows. The Text Screen is also used to run the Logo editor. When the editor is activated the title of the window changes from COMMAND to EDIT. The editor section will cover this in more detail.

From the command window Logo accepts lines of text and acts on them immediately. A line may be typed from the keyboard and then entered with the RETURN key. Pressing RETURN ends the current line and asks Logo to act on what you typed. This is called *command execution*. Logo attempts to execute whatever command you type.

When Logo is ready to accept a command line, it will prompt you with a question mark "?". Following this prompt you type your command. Here is an example:

```
? print "victory
VICTORY
```

Here we typed 'print "victory' and pressed RETURN. Logo then executed the **Print** command and displayed the result on the next line. Try these examples:

```
? show 9 + 1
10

? print [sweet victory!]
SWEET VICTORY!

? show [hello, amiga logo!]
[HELLO, AMIGA LOGO!]

? 37 * 10
370.0
```

What's this? Where's the **Print** command in that last example? There is none. Amiga Logo does not require a **Print** (or a **Show**) for simple commands. (The **Show** command is similar to **Print**, but will print

the outer parentheses of a list.) This arrangement
makes it easier for you to use Logo as a calculator:

```
? 1 + 12 + 4
17

? 3 * 4 * 5
60.0

? sine 30
0.5
```

The results are printed as if Logo had been asked to
do a **Show** command for each line.

Making Logo Talk

Not only can Amiga logo print what you type , it can
speak too. Use the **Say** command like you would the
Print command.

```
? say "hello
```

```
? say [Hello, Your wish is my command]
```

To hear speech, you must have an amplifier or
monitor (with built-in amplifier) connected to the
Amiga audio outputs. The **Say** command will work
only under Workbench 1.3, and the SPEECH device
must be mounted.

Stopping a Command

When you press RETURN, you start a Logo
command executing. Sometimes, you will want to
stop a command before it has finished. In Logo you
do this by holding down the CTRL key while you
press G (CTRL-G we call it). When you do this, Logo
will tell you that it has stopped.

```
? repeat 10000 [print "again]
AGAIN
. . .
AGAIN

. . .
AGAIN
. . .
STOPPED
?
```

Editing a Line

It is convenient to be able to edit a line that has already been typed. The command window of Amiga Logo provides you with all of the editing features of the standard Logo screen editor. You can position the cursor anywhere on the screen, scroll the window up or down, insert and delete text anywhere, copy and move a line, etc. Once you've learned the editing keys, you can use them in both the command and edit windows. Refer to the enclosed reference card for a summary of editing keys.

As you have already noticed, typing a character will print it on the screen at the position of the cursor and move the cursor forward one position. Pressing BACKSPACE will delete the character just typed. There is a restriction though: you can only backspace to the beginning of a line, no further. To delete the character under the cursor, press DELETE or CTRL-D (hold down CTRL and press D). You can only delete characters to the end of a line.

Moving Around

To move backward without deleting a character, use the LEFT-ARROW or CTRL-B. To move forward use RIGHT-ARROW or CTRL-F.

To move to the beginning of a line (even if it wraps around the edge of the screen) press SHIFT-LEFT-ARROW or CTRL-A. To move to the end of a line (the last printed character on a line), use SHIFT-RIGHT-ARROW or CTRL-E.

To move to the previous line, press UP-ARROW or CTRL-P. To move to the next line, press DOWN-ARROW or CTRL-N.

Another way to quickly move the cursor is with the mouse. Position the mouse pointer anywhere on the text screen and press the left mouse button. The cursor will jump to the space nearest the mouse pointer.

Scrolling

To scroll forward a half page, use SHIFT-DOWN-ARROW or CTRL-V. To scroll back a half page, SHIFT-UP-ARROW or CTRL-R. If you want to center your current line on the page, try CTRL-L.

Deleting a Line

To erase all characters from the cursor to the end of a line, type CTRL-K. To delete an entire line, type CTRL-A, CTRL-K. Typing CTRL-K at the end of a line will join the next line to the end of the current line.

Copying and Moving a Line

Use CTRL-K to cut characters and put them into memory. To recover or paste those characters onto the screen, move the cursor to the desired position and type CTRL-Y. CTRL-Y yanks back the latest group cut by CTRL-K.

Splitting a Line

To split a line into two parts, use CTRL-J or CTRL-O. To join two lines, use CTRL-K at the end of the first line.

Re-entering a Line

Amiga Logo will accept a command line from anywhere on the screen. You never need to retype a line. If you move your cursor to a previous line and press RETURN, Amiga Logo will accept that line as if you typed it yourself. This feature makes it easy to experiment and try new ideas without retyping the commands each time. It's also a good way to fix a line after an error.

You need not restrict yourself to just entering previously typed lines. Amiga Logo will accept *any*

text that is on the screen, even the output of previous operations. For example, typing:

```
? print [1024 + 37]
```

would result in:

```
? print [1024 + 37]
1024 + 37
?
```

If you move up a line (UP-ARROW or CTRL-P) and press RETURN, the result would be:

```
? print [1024 + 37]
1024 + 37
1061
?
```

The second line is entered as a command: the numbers are added together and printed.

Long Lines

A line of text can be longer than the width of the screen. When Logo detects such a line, it will automatically wrap it to the beginning of the next line. To indicate that the line is continued, an ! will be printed on the right side of the text window. Long lines can be edited just like any others.

Logo restricts a long line to no more than 240 characters in length. If you require more than this, use a procedure and divide the line up into multiple lines.

Erasing Text

To erase all text in the text window, type "ClearText" or "CT". Everything in the text window will be erased and the text cursor will be positioned to line one. When you execute "ClearText," all of the memory used by text is freed for reuse in Logo.
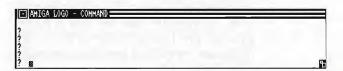
Included with this manual is an Amiga Logo Reference Card. This card includes a summary of the control keys described above.

The Graphics Window

Above the text window, there is another window called the *Graphics* window. This is the window where all your graphic images are drawn. It has a fixed size (it is always the full size of the screen) and it cannot be moved about on the screen.

```
AMIGA LOGO                                            [▓][▓]
```

The Graphics —————————
Window

```
[□] AMIGA LOGO - COMMAND
?
?
?
?
? ▓                                                      [▐]
```

There are many ways to expose the graphics window:

- You can type the command **FullScreen** (or just **FS**) in the command window.

- You can select the *FullScreen* item from the SCREEN menu. If necessary refer to your Amiga user's manual for instructions on using menus.

- You can press CTRL-T. This will flip you back and forth between the fullscreen text and graphics windows.

- You can close your text window by clicking the close box (in the upper left corner) with the left mouse button.

- You can resize your text window by dragging the gadget in the lower right corner of the window.

Any of these methods will work. Find the one that you prefer and practice using it. To reopen the Text window:

- You can type the command **TextScreen** (or just **TS**) and hit RETURN. This may be hard to do because you won't be able to see what you type.

- You can select the *TextScreen* item from the SCREEN menu.

- You can press CTRL-T. This will flip you back and forth between the text and graphics windows.

Mouse Draw

With the graphics window exposed you will be able to watch as Logo draws your images. In addition, you can draw your own images directly with the Amiga mouse. We call this ability *Mouse Draw*.

To draw a line with the mouse, position it within the graphics window and press the left mouse button. While holding the button down, move the mouse and watch what happens. If the pen is in its down position, a line will be drawn using the current pen color (see the graphics tutorial section for more information).
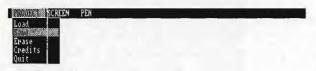
Draw free-hand graphics using the mouse

```
┌─┐AMIGA LOGO - COMMAND
I don't know how to ?CS
? clearscreen
? ht
? st
?
?
```

Load,Save,Erase
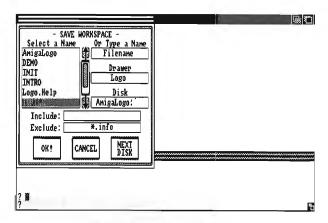
Once you become familiar with Logo, you may want to save your new programs to disk. A saved program can be loaded back into Logo, restoring all your procedures and variables. If you do not save your program it will be lost when you exit Logo or turn off your machine.

To save all procedures and variables that you have created, move the mouse to the top border of the screen, click the right mouse button, and select the *Save* item from the PROJECT menu.

```
PROJECT SCREEN   PEN
 Load
 
 Erase
 Credits
 Quit
```

```
┌─┐AMIGA LOGO - COMMAND


?
```

A requestor will appear. Type the name of your file in the space provided, press RETURN, and click on OK!.

```
- SAVE WORKSPACE -
Select a Name      Or Type a Name
AmigaLogo              Filename
DEMO
INIT                    Drawer
INTRO                    Logo
Logo.Help
                        Disk
                     AmigaLogo:

Include:
Exclude:     *.info

  OK!     CANCEL    NEXT
                    DISK
```

If you would like to save a file from the command window, type:

? save "filename
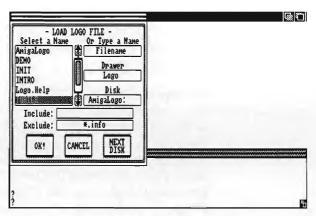
where, filename is the name for your file. Do not forget to put the quote in front of the name (but *do not* put a quote at the end of the name). You will learn more about quotes in the next chapter. Note that the save command saves the program only and not the program mode (such as screen resolution and number of colors). Pictures made with the mouse are saved as screen dumps.

Once a file has been saved, it can be loaded back into Logo in a similar fashion. Select the *Load* item from the PROJECT menu.

SCREEN   PEN

Save
Erase
Credits
Quit

AMIGA LOGO - COMMAND

?
?

Select or type the name of the file to load.

- LOAD LOGO FILE -
Select a Name   Or Type a Name

AmigaLogo   Filename
DEMO
INIT   Drawer
INTRO   Logo
Logo.Help   Disk
                   AmigaLogo:

Include:
Exclude: *.info

OK!   CANCEL   NEXT
                 DISK

?
?

You can also load your file from the command window by typing:

? load "filename
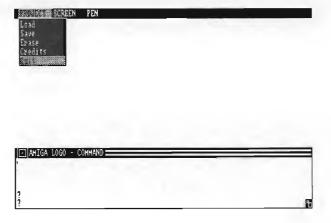
Whenever you run Logo, a start-up file called INI[
will be automatically loaded. This file can contain a
utility procedures or variables that you desire. It c
also execute start-up commands. This file can be
modified with the Logo screen editor and the Editl
command. See the *Mastering Logo* chapter for mor
detail.

It is not always convenient to save all procedures and variables to a single file. Sometime you will want to store a group of related procedures and variables as a single unit. This can be done with *packages*. See the *Mastering Logo* chapter for more detail.

To erase a procedure that you have saved, move the mouse to the top border of the screen, click the right button, and select the **Erase** item from the PROJECT menu. A requestor will appear similar to the load and save requestors. Type the name of the file (or click on its name) press **RETURN,** and click on OK. Note that you CANNOT recover a file that has been erased, so use caution and make sure you erase only the files you want to erase.

Exiting Logo

To exit Logo, select the *Quit* item from the PROJECT menu, or type **Exit** from the command window. Before exiting, don't forget to save any changes that you made (see the **Save** command).

no longer type. Exit from the command window.
Before exiting, run "Temp! k..." to save the changes that
you made (see the Save command).

Elements of Logo

Now that you have some idea of how to use Logo, this chapter will acquaint you with the main components of the Logo programming language. This section will cover primitives, procedures, graphics, words, numbers, lists, variables, properties, flow of control, and external inputs.

Primitives

A *primitive* is a command or operation provided as a built-in feature of Amiga Logo. For example the **Print** and **Show** commands are both primitives. So are + for addition, − for subtraction, * for multiplication, etc. In fact, there are more than 160 system primitives provided in Amiga Logo.

Abbreviations

To make them easier to type, many Logo primitives have an abbreviated spelling. For example most of the graphics commands have two letter abbreviations that mean the same thing:

FD for **Forward**
BK for **Back**
PU for **PenUp**
PD for **PenDown**

So you could type **FD** if you didn't want to type **Forward**, and it would mean exactly the same thing. Unfortunately, it would also be more difficult to read.

Synonyms

Some primitives also have a *synonym* (an alternate spelling that means the same thing). Synonyms are usually provided just for compatibility with other

Logos. For example, the **Scrunch** synonym is the old way of saying **Aspect**. They do exactly the same operation.

Amiga Logo Help

Your Amiga Logo program has built-in help messages. If you do not know or remember the function of a primitive, press the **Help** key (or manually type "Help", quotation marks ("), and the name of the primitive in question.

```
HELP "CLEARSCREEN
CLEARSCREEN is a protected primitive
Clear the graphics screen and home the turtle.
```

Logo will print out a message describing the function and usage of the primitive in question.

Inputs

Many of the system primitives require *inputs* for additional information. These inputs are a way of passing objects the primitives use during their execution. For example the **Forward** command requires one input, a number, to indicate the distance to move the turtle:

```
? forward 37
```

Other primitives will require more than a single input, and some do not need any inputs. The reference part of this manual indicates the number of inputs for each primitive.

Inputs usually follow to the right of the primitive being used. However, for some primitives it is more natural to put the inputs on each side. Logo permits this for a few of the arithmetic primitives: $+ - * / =$ $< > \wedge$. We call these types of primitives *infix* operators. We are used to seeing expressions like:

```
1 + 2
```

rather than:

```
+ 1 2
```

Even so, Logo will accept the inputs either way. The first is easier to read. The second is consistent with how all other primitives are used. It really doesn't matter which you decide to use, *but it is recommended that you don't mix the two together*.

Optional Inputs

A small number of primitives accept an input, but do not require it. This is called an *optional* input. For example you could type

catalog ''devs:

or just

catalog

To understand what happens when an optional input is not provided, you must look up the description for the particular primitive in the reference part of this manual.

Variable Inputs

Some primitives will accept a variable number of inputs, and apply their function to each input provided. To indicate that there are a variable number of inputs, enclose the operation and its inputs in parenthesis. For example **Print** will accept more than one input when typed like this:

? (print ''testing 1 2 3)
TESTING 1 2 3

You can add the first five whole numbers with:

1 + 2 + 3 + 4 + 5

or with

(+ 1 2 3 4 5)

In addition to inputs, some of the system primitives *output* a result when they have finished executing. This output can be passed along as an input to another primitive.

```
? print sine 30
0.5
```

In this example, **Sine** returns its output as the input to **Print**. In this way we can chain several primitives together, passing the output of one to the input of the next.

Primitives that emit an output are often called *operations*. Primitives that do not are just called *commands*.

Optional Outputs

Primitives that execute lists of commands (**Run, Repeat, If,** etc.) may or may not output a result. If the list they execute results in an output, they will pass it back as their own.

Procedures
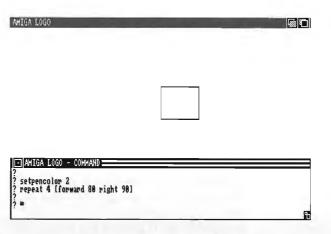
As we described at the start of this chapter, primitives are the built-in commands and operations used to make things happen in Logo. It is also possible and useful for you to create your own commands and operations. These are called *procedures*.

A procedure is a list of command lines that you group together to perform whatever function you desire. When you give this list a name, you are *defining* a procedure.

Suppose you wanted to draw a box of a certain color. These command lines would do the job:

```
setpencolor 2
repeat 4 [forward 80 right 90]
```

```
AMIGA LOGO                                              ▓▓▒
```

```
□ AMIGA LOGO - COMMAND
?
? setpencolor 2
? repeat 4 [forward 80 right 90]
?
? ▪
```

If you wanted to draw the box several times, you would have to re-enter these lines each time. An easier way would be to give these lines a name by making them a procedure:
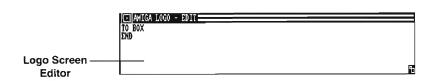
```
to box
setpencolor 2
repeat 4 [forward 80 right 90]
end
```

Here the **To** word tells Logo that you want to define a new procedure called **box**. You could just as easily call it **mybox** or **box80**. It really doesn't matter much to Logo so long as you don't give it the name of a Logo primitive. It is a good idea to give it a name that means something and is easy to read.

If you try typing this example into the command window, something strange will happen. Just as soon

as you've entered the **To box** line, the text window
will clear itself and print:

```
AMIGA LOGO                                              ▨▣
```

```
□ AMIGA LOGO - EDIT
TO BOX
END
```

**Logo Screen
Editor**

and the title of the window will change to EDIT. You
are now in the Logo screen editor. The commands for
moving the cursor are the same as those you've
learned for the command window. However, when
you press RETURN in the editor, the command is not
executed; the cursor just moves to the next line.

The **End** word simply tells Logo where your new
procedure ends. It must be the last line of your Logo
procedure.

Just like the command window, you can move the
cursor around the edit window to correct any
mistakes you've made. When you've finished, exit
the editor and enter your new procedure by pressing
CTRL-C. The command window will return just as
you left it.

To run your **box** procedure, type it as if it were a
command:

? box

To view the results, you may need to expose the
graphics screen by pressing CTRL-T. To return to the
command screen, press CTRL-T again.

Editing a Procedure

Let's say you want to modify your procedure to make it draw a larger box. With the Logo editor, this is simple. Just type:

```
? edit "box
```

Notice that you need to put quotes (") in front of the procedure name. If you didn't do this, Logo would execute the procedure before starting the editor.

If you know that **box** is the last thing you edited, you can return to edit it again by typing:

```
? edit
```

The editor will display:

```
TO BOX
setpencolor 2
repeat 4 [forward 80 right 90]
END
```

Use your cursor keys and the BACKSPACE or DEL key to change 80 to 100. Press CTRL-C. Your modified procedure is now defined.

Inputs

Suppose you wanted to modify your procedure to draw a box of any size. Just like with primitives, procedures can have *inputs*.

To define an input for a procedure, place the name of an input variable in the *title* line. For example:

```
TO BOX :size
```

would tell Logo that you want **box** to accept a single input, and that within the procedure, this input will be accessed with the **size** variable. The actual name of the variable is up to you, but just as with other variables, it's a good idea to give it a name that is easy to read and understand.

With the editor, modify your example to match this:

```
TO BOX :size
setpencolor 2
repeat 4 [forward :size right 90]
END
```

Notice the **size** variable appears in two places: where it is defined as an input, and where it is used as a distance to **Forward**.

Whatever number you now supply as input to **box** will be supplied to **Forward**. This will allow you to draw boxes of various sizes: Try these to create boxes of different sizes:

```
? box 30

? box 46

? box 65
```

Adding another input is similar to adding the first. Let's say we want to add a color input:

```
TO BOX :size :color
setpencolor :color
repeat 4 [forward :size right 90]
END
```

Then we could type:

```
? box 30 1

? box 55 3

? box 72 2
```

Here the second input is the color of the box.

Local Variables

A *local* variable is like any other Logo variable, except that it only exists when Logo is running in the procedure that defined it. The value of a local variable cannot be referenced from outside of its procedure.

There are two types of local variables. You are already familiar with one: the input variable. The other is a variable defined with the **Local** command within a procedure:

```
to grid :size
local "angle
make "angle 10
. . .
end
```

Here both **:size** and **:angle** are local variables in a procedure called **grid**. The **:size** variable gets its value from the input passed to **grid** when it is used. For example:

```
? grid 10
```

will set **:size** to 10. The **:angle** variable gets its value from the **Make** command inside **grid**. Both these values are *private* to the **grid** procedure. They cannot be accessed from outside **grid**.

A variable that is not local is called a *global* variable. Such variables can be accessed from the command level or from any procedure.

Name Conflicts

What happens when a local variable has the same name as a global variable? Logo takes care of this situation by saving the value of the global variable before setting-up the local variable, and it will restore the global's value when the procedure has finished. Thus, the global value will not be affected by the local use of the variable.

This may seem confusing, so an example might help. Create a procedure that changes the value of :a and then prints it:

```
to printit
make "a 2
print :a
end
```

Then it makes sense for these results:

```
? make "a 1
? printit
2
? print :a
2
```

Now edit the procedure to add just one line that makes :a a local variable:

```
to printit
local "a
make "a 2
print :a
end
```

Now try this:

```
? make "a 1
? printit
2
? print :a
1
```

The variable has maintained its global value outside of **printit**. It was equal to 2 only while inside the procedure.

What would the value of **:a** be if we called another procedure within **printit**? Try this:

```
to printagain
print :a
end

to printit
local :a
make :a 2
print :a
printagain
end
```

Now run it:

```
? make ''a 1
? printit
2
2
? print :a
1
```

The value of **:a** in **printagain** remains as it was in **printit**! It turns out that a local variable's value exists within its procedure and *within any procedure invoked by that procedure*.

Output

Procedures, like primitives, can also output values. The **Output** primitive makes this possible:

```
to double :num
output :num + :num
end
```

This procedure would double its input value and output the result:

```
? print double 3
6
```

```
? print double double 4
16
```

The value passed as input to **Output** will be the result of the current procedure, and execution of the procedure will be stopped.

If you simply want to stop a procedure without returning a value, the **Stop** command will do so. No output will be returned.

Building Up

As you become more experienced in Logo, you will build many different procedures. Some of these procedures you will save and reuse in new programs. After a while your procedures will in turn call other procedures, which may call yet other procedures, etc. This is how large, complex programs are written.

Each procedure will eventually depend on the features of other procedures, just as they would the features of the Logo primitives. Once a procedure has been created and debugged, you should think of *what* it does and not *how* it does it. This relieves your mind of trying to remember too many details. In your mind you automatically relate the name to the action. This is called *abstraction.* It is the *essence of good programming.*

You will want to give your procedures good names to reflect what they do. In addition you should minimize the side effects of each procedure—avoid the use of variables that are not local to the procedure. If you do these things, you will rarely need to look at procedure definitions to figure things out.
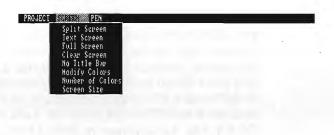
Last of all, make your procedures simple. Opt for a greater number of simple procedures, rather than a lesser number of complex procedures.
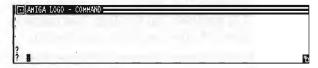
Graphics

One of the primary features of Logo is its ability to easily draw color images on the computer's screen. This section will describe the basic ideas for creating your own images in Logo.

The Screen

As described above in the section *The Graphics Window*, the **FullScreen (FS)** and **SplitScreen (SS)** primitives are both ways to display the graphics portion of the screen. You can return to the full text command and editor windows with **TextScreen (TS)**. These primitives can also be selected in the SCREEN menu bar at the top border of the screen.

```
PROJECT  SCREEN  PEN
         Split Screen
         Text Screen
         Full Screen
         Clear Screen
         No Title Bar
         Modify Colors
         Number of Colors
         Screen Size
```

```
[x] AMIGA LOGO - COMMAND
.
.
?
? ▓                                          ▐b
```
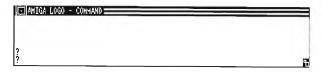
You can also press CTRL-T to switch between text and graphics screens.
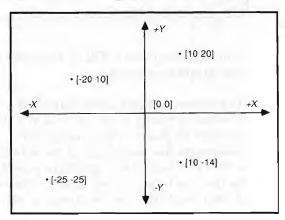
The screen is made up of little dots called *pixels*. The number of pixels in the vertical or Y direction and the number of pixels in the horizontal or X direction determine the resolution of the screen. In turn, the width and height of a graphics image is determined by the resolution of the screen. The most common screen resolution on the Amiga is 640 X 200, which is

also its default resolution. Other possible values
include: 320 X 200, 320 X 400, and 640 X 400. The
screen resolution can be changed with the SCREEN
menu *Screen Size* item.

Points are drawn on the screen using an imaginary
grid. For every point there is a horizontal and vertical
position that can be used to describe it. The
horizontal position is often called the X coordinate,
and the vertical is called the Y coordinate. You need
both X and Y to indicate a position on the screen,
such as X at 10, Y at 20, or in the form of a Logo list
[10 20]. The X coordinate is always first in the list. The
point [0 0] is at the center of the screen. To the right,
X is positive. To the left, X is negative. Y is positive
upward, and negative downward.

[0 0] is called the *origin* or *home* position.

The range of X and Y in each direction depend on the screen resolution that you are using:

Scrn Res.	− X	+ X	− Y	+ Y
320 X 200	− 160	+ 160	− 122	+ 122
320 X 400	− 160	+ 160	− 122	+ 122
640 X 200	− 320	+ 320	− 244	+ 244
640 X 400	− 320	+ 320	− 244	+ 244

The width and height of a graphics image may also be altered by using the **SetAspect** primitive to change the ratio of the number of units in the Y direction for each unit in the X direction. This will be explained in more detail at the end of this section.

The Turtle

Logo's graphics may also be called Turtle Graphics. This is because of a small image that functions as a cursor to let the user know where the pen is. This image is called a turtle and actually looks like a turtle in Amiga Logo. While the pen is down the turtle will draw a line on the graphics screen wherever it is told to go.

The turtle is usually visible on the graphics screen and will move according to the commands typed on the text screen or wherever the mouse goes if the button is pushed down. It rotates so that its head is pointing the direction that it will move next.

The turtle always starts in the same place on the screen when Amiga Logo is started. This place on the screen is called home. When the user wants to place the turtle back to its original position the **Home** primitive will accomplish that without disturbing the rest of the graphics screen. The user can also clear the graphics screen and home the turtle by using the **Clearscreen** primitive. One more option is to clear the graphics screen without moving the turtle by using the **Clean** primitive.

The user may not always want the turtle to be visible.

Logo provides three primitives to give the user flexibility. The **ShowTurtle** primitive causes the turtle to be shown on the graphics screen, and the **HideTurtle** primitive causes the turtle to disappear. The **Shown?** primitive returns a TRUE or FALSE depending on whether the turtle is showing or not.

Here is an example of these primitives:

```
? hideturtle
? shown?
FALSE

? showturtle
? shown?
TRUE
```

The Pen

The concept of a pen is very similar to what you are used to drawing with. This pen is attached to the turtle instead of your hand. The position and direction of the pen are shown by the turtle. The pen can be either up or down. Just like when you draw with your hand if the pen isn't down you don't make any marks. It is important to remember that being able to see the turtle does not reflect the up or down position of the pen.

The pen is put into the down position so it can draw with the **PenDown** command. To put the pen into the up position, use the **PenUp** command. After putting the pen into the up position, the turtle will move around the screen but no lines will be drawn until the pen is put into the down position again.

PenErase will change the pen into an eraser. While set to this, the pen will erase any line or area it moves over. You can stop erasing with **PenUp** or **PenDown**.

You can select any of the above commands by typing them in the command window or by accessing the Pen menu at the top border of the screen and selecting an option with the mouse.
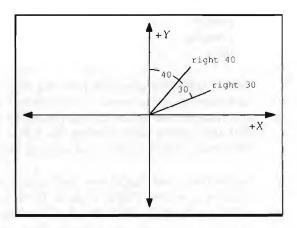
PenReverse will set the pen to reverse any colors that it passes over. On the Amiga, we call this the *complement* mode of graphics. The reverse of a color depends on the number of colors you are using. See the color section below. You can return to normal pen operation with **PenUp** or **PenDown**.

Heading

The direction the turtle is pointing is called its *heading*. This is the angle measured in the clockwise direction from straight up. Headings are always expressed in degrees with zero degrees being straight up.

The heading angle increases as the turtle rotates in the clockwise direction. When the heading hits 360, it is automatically reset to zero.

To turn the turtle to a heading clockwise from its current position use the **Right** command. To turn counterclockwise use the **Left** command. Both of these cause the turtle to rotate *relative* to the current turtle heading. So if the turtle were heading 40 degrees, and you rotated right by 30, its new heading would be 70 degrees.

It is easy to lose track of the exact heading of the turtle. The **Heading** operation will always return the current turtle heading. We know that **Home** faces the turtle up, which is zero degrees:

```
? home
? heading
0
```

You could tell the turtle to rotate 90 degrees to the right with:

```
? right 90
```

and the heading would become:

```
? heading
90.0
```

Doing this again:

```
? right 90
? heading
180.0
```

The turtle now points straight down. To rotate back a bit try:

```
? left 30
? heading
150.0
```

Notice that changing the heading will cause the turtle to rotate on the screen. The turtle will always be pointing in the approximate direction of its heading. But also notice that rotating the turtle does not draw any lines (that's in the next section below).

Remember that **Right** and **Left** rotate the turtle by a relative amount. **Right** adds to the heading. **Left** subtracts from it. With the **SetHeading** primitive you specify an *absolute* angle that will become the new turtle heading.

```
? setheading 230
? heading
230.0
```

One more convenient primitive is **Towards**. If you
know a position on the screen in terms of its X and Y
coordinates, you may want to know what heading
angle points you there. **Towards** will return the
heading required to point in a given direction. From
the home position:

```
? towards [10 10]
45.0

? towards [-30 0]
270.0
```

You can input this angle to **SetHeading**:
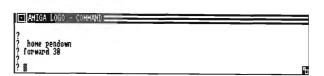
```
? setheading towards [-30 0]
? heading
270.0
```

With **Towards** a new heading can be established
without knowing the angle.

Movement

To actually move the turtle forward you use the
Forward command. This will move the turtle a
specified distance *relative* to its current position. The
turtle will move in the direction of its heading. If the
pen is down, a line will be drawn.

```
? home pendown
? forward 30
```
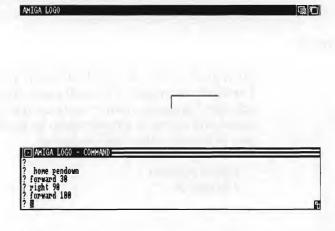
```
AMIGA LOGO - COMMAND
?
?  home pendown
?  forward 30
?
? █
```

The heading is 0 after a Home, so the turtle will move up by 30 units. The pen is down, so a line will be drawn.

    ? right 90
    ? forward 100

Turn right and move forward 100 units.

```
AMIGA LOGO - COMMAND
?
?  home pendown
?  forward 30
?  right 90
?  forward 100
? █
```

Back will move the turtle in the direction opposite the heading:

    ? back 200

In this case, the turtle will move to the left.

```
?  home pendown
?  forward 30
?  right 90
?  forward 100
?  back 200
?  ▓
```

To determine the turtle's position, the **Position**
operation will output a list of the current X and Y
coordinates:

```
? home
? position
[0 0]
```

```
? forward 30
? position
[0 30.0]
```

The position and heading of the turtle are two
different things; do not confuse them. The **Right** and
Left primitives have no effect on the turtle's position,
only its heading.

To place the turtle in a particular location on the
graphics screen use **SetPosition**. This moves the
turtle to an *absolute* position; it doesn't matter where
the turtle came from.

```
? setposition [10.0 20.0]
? position
[10.0 20.0]
```

You can set the X and Y positions separately with
SetX and **SetY**.

```
? setx 40
? position
[40.0 20.0]


? sety 0
? position
[40.0 0.0]
```

It is also possible to obtain the X and Y coordinates separately with **XPos** and **YPos**:

```
? xpos
40.0


? ypos
0
```
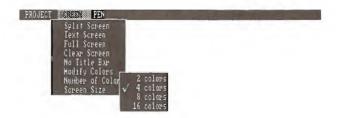
Colors

Color is an essential part of Logo as it enriches the creativity of the user. The Amiga has the capability of displaying 4096 different colors, and Amiga Logo permits the full use of these colors.

The number of colors actually displayed at one time depends on the amount of memory used to hold the graphics image. The more memory used, the more colors displayed.

Amiga Logo can display 2, 4, 8, 16, or 32 colors at the same time. Of course one color is always used for the background, so this leaves 1, 3, 7, 15, and 31 colors for your use. Amiga Logo starts out with 4 colors. The number of colors is set with the *Number of Colors* item in the SCREEN menu.

PROJECT SCREEN PEN

Split Screen
Text Screen
Full Screen
Clear Screen
No Title Bar
Modify Colors
Number of Color   2 colors
Screen Size   ✓ 4 colors
                       8 colors
                      16 colors

AMIGA LOGO - COMMAND
?
?
?
?
?
?

The ability to use 31 colors is also dependent on your screen resolution. If your horizontal resolution is greater than 320, Logo will restrict you to 16 colors at most. You will notice that the menu does not include the 32 color option in such cases.

You may not always have enough memory to display all colors. If you are running other programs on your Amiga at the same time you run Logo, they will decrease the amount of free memory available. If you need to, run Logo alone.

In Logo, colors are assigned to the pen by number. Depending on the number of colors you selected there are 1 to 31 different colors numbers. When Logo starts running it assigns a color to each number. The first 4 initial colors are:

0   *black (background)*
1   *white*
2   *green*
3   *violet*

These numbers are used by both the background and pen to determine their color. Initially, the background color is zero and the pen color is one. Color number zero is a special color that always represents the color of the background. For a complete list of Logo's default colors, turn to p. 172.

To change the color of the background, you can assign a different color number with **SetBackground**:

? setbackground 2

Notice that the entire screen and its borders change color. To determine the color number of the background:

? background
2

To change the color number of the pen, **SetPenColor**:

? setpencolor 3
? pendown forward 20
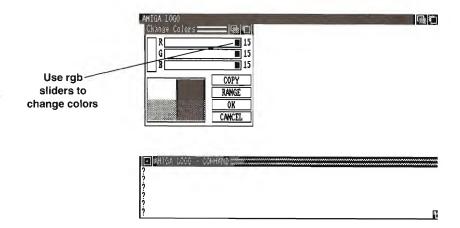
And, to determine the pen color number:

? pencolor
3

On the computer screen all colors are made from three primary colors: red, green, and blue. These colors are mixed in varying quantities to get the desired color. For example, yellow is a mixture of green and red. Purple is a mixture of red and blue. You can mix your own colors by selecting the *Modify Colors* item from the SCREEN menu.

```
PROJECT   SCREEN   PEN
          Split Screen
          Text Screen
          Full Screen
          Clear Screen
          No Title Bar
          Modify Colors
          Number of Colors
          Screen Size
```

Each color can have 0 to 15 units of red, green, or blue. A yellow color might have 14 of red, 13 of green, and no blue. 8 of red, 8 of green, and 8 of blue would make a grey.

AMIGA LOGO
Change Colors
R 15
G 15
B 15
COPY
RANGE
OK
CANCEL

Use rgb sliders to change colors

AMIGA LOGO - COMMAND
?
?
?
?
?
?

With the **SetRGB** command you can change the color displayed by a particular color number (with the exception of color number zero). For example, you might decide to change color number three from purple to yellow:

? setrgb 3 [14 13 0]

The input list is the amount of red, green, and blue (RGB) that you want to mix together to make color number three. Notice that all lines *already drawn* with this color number on the screen change from purple to yellow. You are changing the actual color assigned to the color number.

To obtain the RGB values for an existing color:
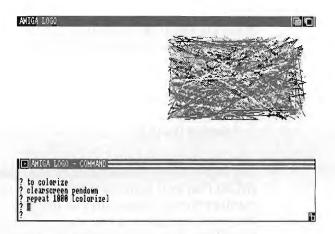
? rgb 1
[14 14 14]

? rgb 2
[0 11 0]

If you modify colors under program control, you can create some interesting effects:

```
to colorize
local ''new.pc
make ''new.pc 1 + random 14
setrgb :new.pc (list random 15 random 15 random 15)
setpencolor :new.pc
setposition list random 300 random 200
end

? clearscreen pendown
? repeat 1000 [colorize]
```



```
? to colorize
? clearscreen pendown
? repeat 1000 [colorize]
?
?
```

Run this example with 16 colors in 640 horizontal resolution.

Filling Areas

Often when creating images it is desirable to make an item a solid color. There are two commands, **Fill** and **FillIn**, which can be used to accomplish this.
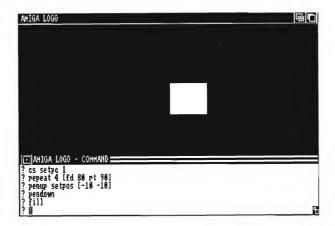
To use **Fill**, draw an object that has a closed border such as a circle or square. Position the turtle inside the object and type **Fill**. The whole object will become the same color as the pen.

```
? cs setpc 1
? repeat 4 [fd 80 rt 90]
? penup setpos [10 10]
? pendown
? fill
```



```
AMIGA LOGO - COMMAND
? cs
? cs setpc 1
? repeat 4 [fd 80 rt 90]
? penup setpos [10 10]
? fill
?
```

The fill will begin at the current pen position, and
flood outward until it hits a pixel on the screen that is
the same color as the pen. If you change the pen
color, you will end up with a different result. Also,
the pen should be down for this command to work
properly.

If you forget to position the turtle inside your object,
the **Fill** would paint the space outside of the object
instead.

```
? cs setpc 1
? repeat 4 [fd 80 rt 90]
? penup setpos [-10 10]
? pendown
? fill
```

```
□ AMIGA LOGO - COMMAND
? cs setpc 1
? repeat 4 [fd 80 rt 90]
? penup setpos [-10 -10]
? pendown
? fill
? ▨                                                            ▣
```

If your object is not completely enclosed, the fill will
leak out and color the whole screen the color of the
pen. When this happens, it may look like the
background color has been changed, but it has simply
been colored over.

Fill colors the area bounded by the current pen color.
It does not matter what other colors are inside this
area they will be washed over. This feature is not true
of the **FillIn** command. **FillIn** will fill to any
boundary, regardless of its color, as long as it is a
different color than the color on which the fill started.
The following example would create a green diamond
with a white border:

```
? cs setpc 1
? repeat 6 [fd 60 rt 60]
? penup setpos [10 10]
? setpc 2
? pendown
? fillin
```

```
□ AMIGA LOGO - COMMAND
? repeat 6 [fd 60 rt 60]
? penup setpos [10 10]
? setpc 2
? pendown
? fillin
?
```

Another way to access the **Fill** and **FillIn** commands is by using the mouse. Click the right mouse button at the top border of the screen to access the Pen menu, and select the item of your choice.

```
PROJECT  SCREEN
              PenUp
              PenDown
              PenErase
              PenReverse
              SetPenColor
              Fill
              FillIn
```

```
□ AMIGA LOGO - COMMAND
? cs
?
?
?
?
```

Turtle Bounds

The turtle can move freely around the graphics screen, but what happens when it tries to cross the edge? It depends. You have three choices.

The **Fence** command directs Logo to limit the turtle to the screen only. If the turtle tries to go beyond the edge, an error message will be generated, and the turtle will be halted at the edge of the screen as if it were fenced in.

```
? home fence
? forward 250
Turtle out of bounds
```

The **Wrap** command directs Logo to let the turtle
move off the screen on any side and reappear on the
opposite side as though the edges were connected.

```
? home wrap
? forward 450
? position
[0.0 -39.08397]
```

The **Window** command specifies that the turtle can
move off the screen into an imaginary space.

```
? home window
? forward 1000
? right 90 forward 2000
? position
[2000.0 1000.0]
```

Fence is the initial mode when Logo is started.

Aspect Ratio

Amiga Logo has a built-in *Aspect* correction for
different screen resolutions. If this were not true, on a
640 X 200 screen a vertical line would seem nearly
twice as long as a horizontal line of the same length.
This distortion is caused by the difference in the
number of pixels vertically when compared to the
number of pixels horizontally. To produce accurate
images, where a square looks like a square and not a
rectangle, Logo adjusts the *aspect ratio*. This ratio
controls the scale of the Y axis compared with the X
axis.

The **Aspect** operation returns the current aspect ratio
of the screen. This is the number of units in the Y
direction for each unit in the X direction. Normally it
is set to 1:

```
? aspect
1.0
```

To set the aspect ratio to a new value, use **SetAspect**.

? setaspect 0.5

This will make each vertical unit half the size of a horizontal unit.
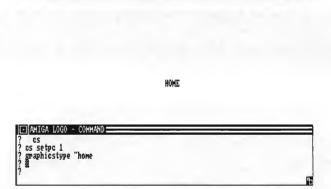
SetAspect helps compensate for different screen resolutions and monitor brands. If your circles look like ellipses and squares look like rectangles, the aspect ratio can correct these.

Drawing Text

We have discussed how to draw lines on the screen, but what about text? The **GraphicsType** command will print its input on the graphics screen as text.

? cs setpc 1
? graphicstype "home

Text will be printed at the current turtle position but the pen position will not change.

The text will be printed in the current pen color and will be affected by the pen's state (up, down, erase, and reverse) just like when drawing a line.

? penup setpos [-57 20] penreverse
? graphicstype [there's no place like]
? penerase graphicstype [there's]
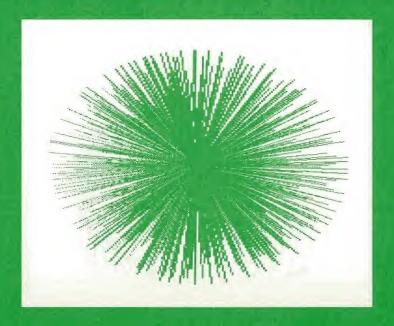
Graphics Projects

Here are some more procedures that generate interesting Logo graphics. Try typing these procedures, saving them to disk, and experimenting with different inputs to achieve different effects.

Explode

cs explode 5
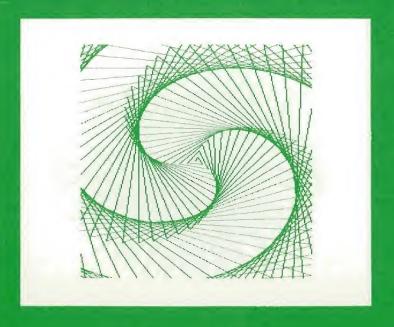

```
TO EXPLODE :SIZE
CS
IF :SIZE>200 [STOP]
HT
REPEAT 36[FD :SIZE BK :SIZE RT RANDOM 10]
SETPC RANDOM8
EXPLODE :SIZE + 5
END
```

Execute this procedure in 640 × 200 resolution with 8 screen colors. You may want to change the procedure by altering the inputs.

Spiro

cs spiro 4 122 200


```
TO SPIRO :SIZE :ANGLE :NUMBER
IF :NUMBER = 0 [STOP]
FD :SIZE
RT :ANGLE
SPIRO (:SIZE + 10) :ANGLE (:NUMBER − 1)
END
```
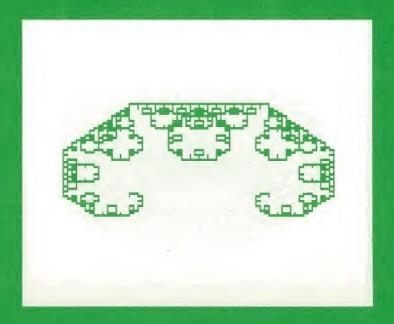
Run this program in window mode. Have fun experimenting with this procedure; you have three inputs to play with. Notice how the graphic changes as you increase the size of the angle.

C

cs c 3 10


```
TO C :LEN :LEVEL
IF :LEVEL = 0 [FD :LEN STOP]
C :LEN :LEVEL − 1
RT 90
C :LEN :LEVEL − 1
LT 90
END
```
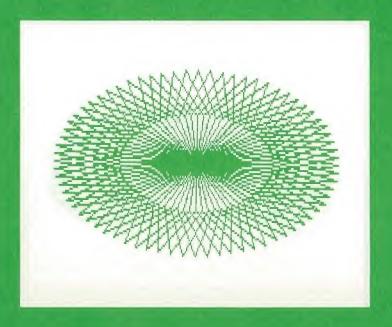
Run this program in window mode to prevent out of bounds error messages from appearing. Notice how increasing the level number makes the graphic more complex.

Sunburst

cs sunburst
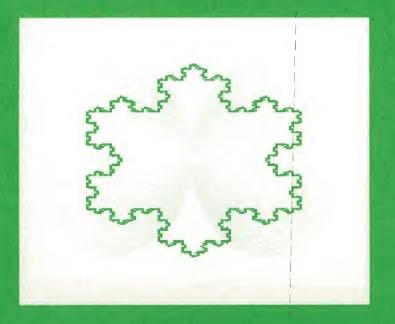

```
TO FOURSIDE
REPEAT 2 [FD 60 RT 30 FD 60 RT 150]
END

TO SUNBURST
HT
REPEAT 72 [FOURSIDE RT 5]
END
```

Notice how this graphic is composed of two
procedures: FOURSIDE and SUNBURST.

Snowflake

cs snowflake 200 5


```
TO SNOWFLAKE :LEN :LEVEL
REPEAT 3 [SIDE :LEN :LEVEL RT 120]
END

TO SIDE :LEN :LEVEL
IF :LEVEL = 0 [FD :LEN STOP]
SIDE :LEN / 3 :LEVEL – 1
LT 60
SIDE :LEN / 3 :LEVEL – 1
RT 120
SIDE :LEN / 3 :LEVEL – 1
LT 60
SIDE :LEN / 3 :LEVEL – 1
END
```
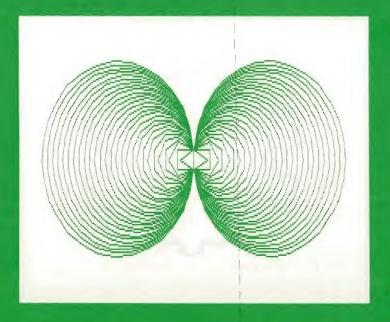
The snowflake graphic is composed of two
procedures: SNOWFLAKE and SIDE.

Polygon

cs polygon 3 34


```
TO POLYGON :SIDES :LENGTH
HT
IF :SIDES = 30 [STOP]
PD
REPEAT :SIDES [FD :LENGTH RT (360/:SIDES)]
PU HOME PD
REPEAT :SIDES [FD :LENGTH LT (360/:SIDES)]
POLYGON :SIDES + 1 :LENGTH
ST
END
```

Run this program in window mode to prevent out of
bounds error messages from occurring. Notice how
the polygons approximate the shape of a circle as
they grow larger.

Fan
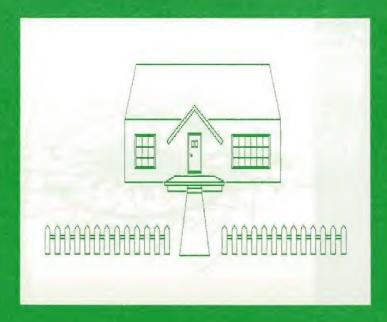
cs fan


```
TO FAN
PU
RT 20
PD REPEAT 3[RARC 50 60 LARC 50 90 BK 50 LT 90]
FAN
END

TO LARC :RAD :DEG
REPEAT 0.0174603 * :DEG * :RAD [FD 1 LT 57.27273 / :RAD]
END

TO RARC :RAD :DEG
REPEAT 0.0174603 * :DEG * :RAD[FD 1 RT 57.27273 / :RAD]
END
```

House

The procedures for the **HOUSE** graphic are too long
to print in the manual; however, you may view them
by double clicking on the **HOUSE** icon in the Logo
drawer of your Amiga Logo system disk. Notice how
a number of procedures have been packaged together
within the main procedure.

Car

To view the procedures for the **CAR** graphic, double
click on the **CAR** icon in the Logo drawer of your
AmigaLogo disk, and type EDIT "CAR" in the
command window. Notice how each procedure
draws a different part of the car.

Words are the primary data element used in Logo. Words are used for numbers, for symbols, and for the names of variables, primitives, and procedures. Several words can be combined to form a sentence or a list.

A word is simply a sequence of characters. For example

plum

A1

star.fish

X32Y70

victory!

what?

37

3.14159

are all words.

When a word is preceded by a quote (") character, it is a *literal* word. Logo will not attempt to do anything special with the word. It will accept it literally as just a collection of letters. Notice that only a starting quote is needed.

```
? print "plum
PLUM

? "what?
WHAT?
```

If a quote is not present, Logo will interpret the word as if it were a primitive or procedure. If there is no such thing, an error will occur:

```
? print plum
I don't know how to PLUM
```

Numbers do not require a quote. They will always be interpreted as words.

```
? print "123
123

? print 123
123
```

Special Delimiters

A word normally ends with a space. The space is said to *delimit* the end of the word. There are other delimiters that also will end the word. They are + − * / = < > [ ] ( ) and ^ . Occasionally you may want to use one of these characters in a word. To do so, you must precede the character with a backslash \ . This tells Logo that the next character should be accepted as part of the word.

```
? print "time\ − out
TIME-OUT

? print "a\*b
A*B

? editfile "s:startup\ − sequence
? print "\[
[
```

However, the \ is not needed if the special character is the first in the the word.

```
? print "*here
*HERE

? catalog "*.info
```

Blank space can also be put into a word using these same rules. For example, to print an empty line type a quote followed by a space:

? print ''

?

Notice the space between the two prompt lines.

Words as Values

Most values in Logo are words or lists of words. Some words have special meaning to Logo. For example, words made up of digits 0 through 9 are numbers. The words "TRUE and "FALSE are special words to indicate logical truth. You can even give a word its own special meaning. You might want to use the word "INVALID to indicate an improper input or such. The word "INVALID does not need to be anything special. It does not need to be a variable or a procedure. It simply represents a value. You can write your own procedure that usually outputs a number, but in an error case might output "WRONG.

Words are also commonly used as strings of characters. In this sense they are similar to strings used in other languages. You would use Logo words to communicate with the user on the screen:

? print ''hello
HELLO

? print [enter your name]
ENTER YOUR NAME

Words as Names

Words are more powerful than just representing values or providing strings of characters. In Logo a word can be the *name* for something.

A word can be the name of a variable, procedure, or primitive. A word can be manipulated as a series of characters, sliced apart and recombined in many

ways, and the result can be used to reference a variable or execute a procedure. *This ability gives Logo power beyond most other programming languages!*

Logo will discern the intended use of a word by the punctuation you provide. Words may be quoted (with a "), unquoted, or dotted (preceded with a colon ":").

You have learned that quoted words and numeric words are literal values:

"victory "cat "true 737 52.5

Unquoted words invoke procedures and primitives (unless the words are within a list — see the section on lists below):

Forward Back Print . . .

Dotted words reference variables:

:color :size :length

Operations

Logo supplies a number of word operations. A word can be broken down into separate pieces (which are themselves words), or several words can be combined to make a new word.

The **First** operation will return the first character of a word:

? first "logo
L

To get the rest of the word **ButFirst** does the job:

? butfirst "logo
OGO

Last and **ButLast** work in a similar fashion on the last character of a word:

```
? last "logo
O

? butlast "logo
LOG
```

To get a specific character in the word:

```
? item 3 "logo
G
```

To count the number of letters in a word:

```
? count "logo
4
```

New words can be created with **FirstPut** and **LastPut**:

```
? firstput "l "ogo
LOGO

? firstput "amiga "logo
AMIGALOGO

? firstput 1 234
1234

? lastput "s "friend
FRIENDS

? lastput "mania "amiga
AMIGAMANIA
```

The **Word** operation is similar to **FirstPut**, but with the use of parentheses, any number of words can be joined:

```
? word "news "letter
NEWSLETTER
```

```
? (word "PORK "U "PINE)
PORKUPINE

? (word "anti "dis "establish "ment)
ANTIDISESTABLISHMENT
```

There are also a number of predicates to use with words. They output either TRUE or FALSE.

```
? empty? "victory
FALSE

? word? "victory
TRUE
```

Two words are equal only if they contain the same characters in the same order.

```
? equal? "victory "theater
FALSE

? equal? "abc word "a "bc
TRUE
```

Another predicate will determine whether a word contains a particular character:

```
? member? "Z "ZIP
TRUE
```

Numbers

In Logo a *number* is a special type of word made up of the digits 0 through 9, and often may include a decimal point, minus sign, or exponent indicator.

Numbers in Logo may be expressed in various ways. For example:

37

10000

-973

are *integer* numbers, whereas

37.45

-45.37

0.75

34.0

.62

8300.

are *decimal* numbers. Notice that integers do not include a decimal point. Also, negative numbers should have no space after the minus sign (or Logo will think that you want to perform an operation like subtraction or negation).

Very large or very small numbers can be represented in a *exponential* form that is similar to scientific notation. For example:

3.7E8

-3.7E5

3.7N4

are just a ways of saying:

370000000.0

-370000.0

0.00037

The letters "E" and "N" are used to indicate the exponent, which is ten raised to the power of the number entered. The "N" indicates a negative exponent.

For the most part Logo treats all types of numbers alike. You can mix and match types as you desire. When it is necessary, Logo will automatically convert a number from one type to the other. For example Logo would convert the input to the **Sine** primitive:

```
? sine 30
0.5
```

from a 30 to 30.0 before performing the operation.

In Amiga Logo numbers have about eight digits of precision. Exponents can range from +18 to -20. When a series of calculations exceeds the precision, the resulting number will not be exact. You may see this occurring from time to time:

```
? 876543210 / 5 * 5 / 876543210
1.0

? (876543210 / 5) * (5 / 876543210)
0.9999999
```

Keep this in mind, as it could create a problem when comparing two numbers for equality.

Operations

Logo provides many built-in operations for numbers: addition, subtraction, multiplication, division, trigometric functions, comparison, random, etc.

As mentioned above in the section about primitives, many of the arithmetic operations are more natural when expressed with the operator between its inputs (rather than in front of them). For example:

    ? 12 + 3

seems more natural than

    + 12 3

It really doesn't matter which one you use—just don't mix the two together; it might be confusing.

Addition is performed with either + or **Sum**. It normally takes two inputs: the numbers to be added.

    ? 120 + 5
    125

    ? 3.25 + 60.2
    63.45

    ? sum 3   52.3
    55.3

Many of the number primitives will accept a variable number of inputs. To indicate the this, enclose the operation and its inputs in parenthesis.

    (+ 1 2 3 4 5)

    (sum 102.3   34.5   76 − 29.46)

Subtraction (−), multiplication (*), and division (/) operate in a similar fashion to addition:

    ? 10 − 3
    7

    ? 10 * 3.3
    33.0

```
? 12 / 4
3.0

? (* 2 3 4 5)
120.0
```

Care must be taken when specifying subtraction.
*There must be a space between the minus sign and the
number being subtracted.* If you forget to put a space,
Logo will think you are just indicating a negative
number:

```
? (print 10 − 3)
```

will print a 7, but:

```
? (print 10 −3)
```

will print a 10 and a −3. Logo thinks that the −3 is a
numeric value, and it will not treat the minus as an
operation.

The minus sign can also be used with a single input to
determine the negative value of a number. Again, to
indicate that you intend an operation, separate the
minus from the input with a space:

```
? print -60
-60

? print - cosine 60
-0.5
```

Amiga Logo supports the common trigometric
functions:

```
? sine 30
0.5

? cosine -45
0.7071069

? tangent 27
0.5095255
```

```
? arctangent 100
89.42706
```

There are operations to compare numbers. You can determine whether a number is greater than, less than, or equal to another number. These operations are predicates and only output a TRUE or FALSE value.

```
? 10 > 3
TRUE

? 13 = 7
FALSE

? 10.3 < 10.1
FALSE
```

If you want to test for a number being greater than or equal, you would need to write an expression like:

```
? not (10 < 0)
TRUE
```

This would be equivalent to saying: return TRUE if 10 is not less than zero. In other words, return TRUE if 10 is greater than or equal to zero. Similarly:

```
? not (100 > 200)
TRUE

? not (32.5 = 17.4)
TRUE
```

would represent less than or equal and not equal.

There is also a predicate to let you test if an input value is a number:

```
? number? 10
TRUE

? number? ''hello
FALSE
```

You can generate random numbers with the **Random** operation:

```
? random 10
7

? random 1000
376
```

(Your output in these examples will differ because the numbers are random!) The input indicates the limit to the random number produced. All your numbers will be less than this limit.

Precedence

When performing a calculation, you will often want to mix several operations within the same line. Normally Logo expressions are evaluated from the left to the right. However, arithmetic operations may be evaluated differently. The multiplication (*) and division (/) operations will be evaluated before addition (+) and subtraction (-). The * and / have a higher *precedence*:

```
? print 3 * 4 + 5
17.0

? print 5 + 3 * 4
17.0
```

Notice that both expressions perform the * first. The result is 17 not 32 (8 * 4).

You can force Logo to evaluate expressions in a different order by grouping your expressions with parentheses.

```
? print (5 + 3) * 4
32.0
```

Logo evaluates expressions in parentheses first. This can be very important in calculations like:

```
? (cosine 10) * 6
5.908847

? cosine (10 * 6)
0.5
```

Lists

In Logo a *list* is a collection of objects. These objects may be words (numbers included) or other lists. Lists are a convenient way to group related values together as a single object. Logo uses lists extensively for even its own internal operations.

When typing and printing lists, square brackets [ ] are used to indicate what elements are included in the list. Everything enclosed by brackets becomes part of the list. Individual elements of a list are separated by a space.

```
[toby]

[a b c d e]

[37 -54 72.4]

[address [100 main street] ]

[ [yellow banana] [green grape] [red apple] ]

[ ]
```

Like literal words, these are *literal lists*. They contain just literal objects; everything within them is a literal. Their contents are not evaluated. Words within literal lists do not need to be quoted.

A list that contains valid Logo commands can be executed. Such a list is called a *run-list*. For example

```
[forward 40 right 90]
[print "tobina]
```

Several Logo primitives accept run-lists as input. They are essential to writing programs in Logo and will be discussed in a later section.

Primitives

Logo supplies several list operations. The individual elements of a list can be extracted, new elements can be added to a list, or several lists can be combined into one.

The **First** operation will return the first element of a list:

```
? first [peach fig cherry]
PEACH
```

To get the rest of the list use **ButFirst**:

```
? butfirst [peach fig cherry]
[FIG CHERRY]
```

Notice that the output is itself a list.

Last and **ButLast** work in a similar fashion:

```
? last [peach fig cherry]
CHERRY
```

```
? butlast [peach fig cherry]
[PEACH FIG]
```

To get a specific element of the list:

```
? item 2 [peach fig cherry]
FIG
```

```
? item 3 [ [yellow banana] [green grape] [red apple] ]
[RED APPLE]
```

To count the number of elements in a list:

? count [peach fig cherry]
3

? count [[yellow banana] [green grape] [red apple]]
3

The **List** operation provides a way to create a new list by joining input objects.

? list 12 "trains
[12 TRAINS]

? list "gems [diamond ruby emerald]
[GEMS [DIAMOND RUBY EMERALD]]

? list [red crab] [gray whale]
[[RED CRAB] [GRAY WHALE]]

When enclosed in parentheses, this operation will join any number of inputs:

? (list 4 "little "yellow "bugs)
[4 LITTLE YELLOW BUGS]

? (list [red crab] [gray whale] [green mac])
[[RED CRAB] [GRAY WHALE] [GREEN MAC]]

Sentence is an operation similar to **List**. It joins input objects together, but will strip the outer brackets off of any lists that may have been input.

? sentence [red crab] [blue whale]
[RED CRAB BLUE WHALE]

? (sentence [there are] 4 "inputs "here)
[THERE ARE 4 INPUTS HERE]

FirstPut and **LastPut** will add elements to the beginning and end of a list:

? firstput "lion [tiger cheetah]
[LION TIGER CHEETAH]

```
? firstput [32 74] [23 89]
[ [32 74] 23 89]

? lastput "keyboard [computer display mouse]
[COMPUTER DISPLAY MOUSE KEYBOARD]
```

There are also a number of predicates to use with lists. They output either TRUE or FALSE.

```
? empty? [cindy sean]
FALSE

? empty? [   ]
TRUE

? list? [cindy sean]
TRUE

? list? [carl]
TRUE

? list? 45
FALSE
```

Two lists are equal only if all of their elements are equal.

```
? equal? [a b] [d e]
FALSE

? equal? [a b] list "a "b
TRUE
```

Another predicate will determine whether a list contains a particular element:

```
? member? "toast [pancake egg bacon toast]
TRUE
```

When a list is printed, its outermost brackets may or may not be shown, depending on the command used. The **Print** command will not print the outer brackets. The **Show** command will.

```
? print [good morning, sean]
GOOD MORNING, SEAN

? show [orange peach]
[ORANGE PEACH]
```

Use the command that makes the most sense for what you are trying to print.

Variables

A *variable* is used to name a value. That is, a variable lets you refer to a value through the use of a name, rather than directly as a constant. This ability turns out to be of prime importance in computing. For example, the command:

```
? name 10 "ten
```

would bind the word "ten" to the number 10.

Another, more common way to create a variable name for a value is with the **Make** command:

```
? make "ten 10
```

Make is identical to **Name** except that the order of the inputs is reversed.

We can now refer to this number with its name "ten", rather than its value. To access the value we ask Logo for the "thing" associated with a name:

```
? print thing "ten
10
```

Here **Thing** is a primitive that outputs to **Print** the value of a variable. It turns out that this operation is so common that Logo provides a shorthand way of obtaining the value of a variable. By placing a colon ":" in front of a variable's name, we refer to its value:

```
? print :ten
10
```

Of course, this variable is not very practical because we normally think of 10 as being ten. But, suppose we wanted to give names to the various colors used in Logo graphics? Logo refers to its colors as numbers. To help us remember the colors, we might give them names:

```
? make "black 0
? make "white 1
? make "green 2
? make "purple 3
```

We could then set the color of the Logo drawing pen with:

```
? setpencolor :green
```

So we no longer need to remember what number represents green.

Of course, the values named with variables don't need to be just numbers. A name can be given to any type of Logo object: words, numbers, lists, and procedures. Say we want to remember the name of your favorite fruit:

```
? make "favorite.fruit "lemon
```

or a list of fruits:

```
? make "fruits [apple lemon orange banana]
```

So far our examples have shown variables with constant values, but this is not the only way of using

variables. As the name suggests, the value of a variable can actually vary. This means we can change the value of something without changing its name.

For example, if your favorite fruit were no longer a lemon, you could change the value of the variable to indicate this:

```
? make "favorite.fruit "plum
```

There is no limit to the number of times the value of a variable can be changed. This means that a variable can be used to save the current *state* of something:

```
? make "num 1
? repeat 4 [print :num make "num :num + 1]
1
2
3
4
```

Here the value of the **num** variable gets changed each time we execute the list. Its new value is created by adding one to its previous value, so the **num** variable indicates the number of times we have executed the list.

Properties

A *property* is a special type of value that can be attached to a Logo word. The properties of a word have names and values, just as variables have names and values.

Any number of arbitrary properties (up to the limits of memory) can be associated with a given word. This makes properties useful as a form of *data base*. With properties it is easy to store and retrieve useful information about an object. For example, let's say you want to store information about various trees.

You could start by defining the properties of a
Redwood tree:

? putprop "redwood "height 350

? putprop "redwood "color "red

? putprop "redwood "location [California Coast]

? putprop "redwood "leaves "evergreen

Here the **PutProp** primitive lets you assign tree
properties and their values to the Redwood tree. The
first input to **PutProp** is the name of the object
(**redwood**) to which the property is attached. The
second input is the name of the property (**height**,
color, **location**, etc.), and the third is the property
value (**350, "red, "evergreen**, etc.). As you see, the
value can be any type of object: number, word, list,
etc.

You could define similar properties for other trees:
maple, oak, fir, etc.

Once the tree properties are defined, you can retrieve
their values with **GetProp**:

? getprop "redwood "location
[CALIFORNIA COAST]

? getprop "redwood "height
350

If you want to remove a particular property there is a
primitive called **RemProp**:

? remprop "redwood "leaves

All other properties will remain intact.

PropList will output a list containing all properties and property values for a word:

```
? proplist "redwood
[LOCATION [CALIFORNIA COAST] COLOR RED HEIGHT 350]
```

Don't depend on the order of the properties in this list. They may change as the list is modified. Notice that they are in a different order than how they were entered.

If you wanted to change an existing property use **PutProp** without removing the property:

```
? putprop "redwood "color [red brown]
```

This will replace the redwood's color property value with [**red brown**]. The old value is lost.

Flow of Control

Commands in a Logo program are normally executed one after another. Each command is read and executed, proceeding through the whole program. This sequence of command execution is called the *flow of control*. It is the order in which your commands are executed. Like most other computer languages, Logo has commands to redirect the flow of control. Much of the power of computers stems from this ability.

Run

We have seen lists used to hold values such as words, numbers, and other lists. In Logo, lists can also be executed. For example the **Run** command will execute whatever list is given to it as input:

```
? run [print "running]
RUNNING
```

At first this doesn't seem too useful, but stop for a moment and think about the possibilities. With list

operations like **List**, **First**, **FirstPut**, **Last**, **LastPut**, etc., you can construct new lists from other Logo objects. Instead of thinking of lists as input data to other procedures, view them as executable commands.

```
? run list "print [victory theater]
VICTORY THEATER

? run lastput [what where when who] [print]
WHAT WHERE WHEN WHO
```

Run is more than just a command. When the last thing it executes is an operation, **Run** acts like an operation and outputs a result:

```
? run (list first [+  −  * /] 2 item 2 [13 56.4 37])
58.4
```

Here is an example of using **Run** to create a procedure that selects between one of many lists to execute depending on a number input:

```
to select :selector :lists
if :selector > count :lists [stop]
if :selector < 1 [stop]
run item :selector :lists
end
```

Here's an example of the **select** procedure at work:

```
? select 2 [ [type "hello] [print "hi] [show "greetings] ]
HI

? select 3 [ [penup] [pendown] [penerase] [penreverse] ]
```

This procedure is programmed to ignore invalid numbers.

When programming you will often want to execute a set of commands several times. The Logo **Repeat** command provides an easy way to do this. Just like the Run command, it accepts an executable list as input. It also accepts an integer that specifies the number of times to execute this list.

For example, a square has four equal sides. To draw one, you could type four identical lines:

```
forward 50 right 90
forward 50 right 90
forward 50 right 90
forward 50 right 90
```

or you could use **Repeat**:

```
repeat 4 [forward 50 right 90]
```

If you wanted to create a procedure to draw any regular polygon:

```
to polygon :sides
repeat :sides [forward 30 right (360 / :sides)]
end
```

so

```
? polygon 4
```

would draw a square, and

```
? polygon 3
```

would draw a triangle.

Occasionally, you may want to keep track of how many times you have repeated the execution of a list. This can be accomplished with a variable:

```
? make "cnt 1
? repeat 3 [print :cnt make "cnt 1 + :cnt]
1
2
3
```

Each time you repeat, the **cnt** variable gets bound to a new value which is one greater than its old value. If you use this technique, remember to set the initial value each time, or you might be in for a surprise.

If you utilize the power of Logo, you could think up a better way to do this. Try this:

```
to thru :var :cnt :do
local :var
make :var 1
repeat :cnt [run :do make :var 1 + thing :var]
end
```

With this new **thru** command, all of the details are handled for you. You can now concentrate on what you want to get done and not worry about how to do it.

```
? thru "i 3 [type :i type " show rgb :i]
1 [14 14 14]
2 [0 11 0]
3 [10 0 10]
```

Another procedure called **for** is similar to **thru**, but the starting number can be specified:

```
to for :var :lo :hi :do
if :hi < :lo [stop]
local :var
make :var :lo
repeat 1 + :hi - :lo [run :do make :var 1 + thing :var]
end
```

So, counting from -2 to 3 would be:

```
? for "i -2 3 [print :i]
-2
-1
0
1.0
2.0
3.0
```

With **thru** you could create a procedure called **foreach** that executes a command using each element from a list as input:

```
to foreach :lst :proc
thru "n count :lst [run se :proc item :n :lst]
end
```

When the input to **thru** executes, **Se** (**Sentence**) will build yet another list to execute, and it is this list that is executed with **Run** to generate the final result.

```
? foreach ["peach "pear "plum] "print
PEACH
PEAR
PLUM
```

Conditional
Execution

Earlier in this chapter we introduced you to the idea of predicate primitives that return only **True** or **False**. The output of a predicate can be the input to a primitive which will control the flow of your program. You can choose to have commands executed only if certain conditions are true or false.

The work horse of these control primitives is the **If** command. It accepts a condition, and either one or two lists as input. If the condition is **True**, the first list is executed. If the condition is **False**, the first list will be skipped, and the second list (if present) will be executed.

```
? if 1 < 2 [print "less] [print "greater]
LESS

? if "true [print "hi]
HI

? showturtle
? if shown? [print [turtle is visible] ]
TURTLE IS VISIBLE
```

If the condition is FALSE and there is no second list, nothing happens.

Notice that the second and third inputs must be a list. If you forget this, Logo will evaluate these inputs and expect their results to be a list. This may cause a rather confusing error message to occur. For example:

```
? if 2 > 1 print "greater
GREATER
PRINT did not output to IF
```

Can you spot what has happened here? The **Print** command was executed, and its result was input to the **If** (which was expecting a list to execute). Because **Print** does not supply an output to **If**, the error occurs. This is a very important and powerful property of Logo. You could, for instance, supply a variable as the list to execute:

```
? make "then.list [print "greater]
? if 2 > 1 :then.list
GREATER
```

Here, Logo evaluates the variable and inputs the resulting list to the **If** command.

Other languages often provide a "while" command which will continually execute a list while a condition remains TRUE. With the recursion available in Logo,

such a function is not required often, but you can
always define it yourself:

```
to while :condition :do
test run :condition
if false [stop]
run :do
while :condition :do
end
```

The inputs are two lists named **condition** and **do**. The
first list will be executed to determine the condition.
If the condition is FALSE, the procedure will stop. If
the condition is TRUE, the second list will be
executed, and the **while** procedure will be executed
again. This technique of a procedure calling itself is
called *recursion*. Here the last command is a call to
itself. This is called *tail recursion* and is used
frequently in Logo.

The line

```
? while [not button? 0] [print "up]
```

will print UP until the mouse button is pressed.

Catch and Throw

Catch and **Throw** are special control primitives that
let you pass control back to a previous point in your
program.

Catch is used to handle *exceptions* to the normal flow
of control in your program. Exceptions are generated
with the **Throw** command or as the result of a
program error.

The first input to both **Catch** and **Throw** is a name. To
transfer control back to a particular **Catch**, its name
must match that used in the **Throw**.

The second input to **Catch** is a list to execute. If it
encounters a **Throw** during the list's execution,
control returns to the line following the **Catch**.

A simple example would be:

```
to catch.x
cs
catch "bounds [repeat 10 [move.x 20]]
print pos
end

to move.x :x
setx (xpos + :x)
if or (xpos > 80) (xpos < -80) [throw "bounds]
end

? catch.x
100.0 0
```

In this example, **Throw** sends control back to the **Catch** when x exceeds 80. This happens before the **Repeat** has finished all ten repetitions.

The **Catch** command must occur before the **Throw** or you will receive an error message from Logo.

```
? throw "bad.apples
Cannot find catch for BAD.APPLES
```

With the **Catch** command errors can be trapped by a program before they are printed and processed by Logo. This is done by using the name **error** for your **Catch:**

```
catch "error [repeat 4 [fd 100 rt 90] ]
```

within one of your procedures. When an error occurs, control will be transferred back to this command *before an error message is generated*. You can then use the **Error** primitive to determine what error occurred. See **Error** for more information.

With **Throw** control can be passed from your procedure, all the way back to the top level (command level) of Logo:

```
throw "toplevel
```

This is provided for compatibility with previous Logos. In most cases it is easier to just invoke the **TopLevel** primitive:

    toplevel

to return to the top level of Logo.

If you ever want to delay program execution for a few moments, the **Wait** primitive will be of interest. It will cause Logo to stop executing for a given period of time. This period is measured in sixtieths of a second (1/60), so typing:

    ? wait 60

will delay for one second.

On the Amiga, this method of waiting is much preferred to other ways of "spinning your wheels" where you might execute a command over and over. The Amiga is multitasking, and it can be doing other things while your program is waiting. The **Wait** command lets it do so.

The **Wait** command also accepts the special word "**Frame** as input. When this is done, Logo will wait until the video circuitry has started its retrace (blanking) before continuing. Using **Frame** helps prevent many of the flicker effects that occur when drawing to the screen. Compare

    ? showturtle home
    ? repeat 90 [right 23]

with

    ? showturtle home
    ? repeat 90 [right 23 wait "frame]

The second will appear to rotate the turtle in a much smoother fashion.

When designing interactive programs it is often necessary to read inputs from the keyboard or mouse. Logo provides a few operations to help you do this.

The **ReadChar** operation will return the next character ready from the keyboard. If a character has already been pressed, it will return immediately, otherwise it will wait until a key is pressed. When executed from a procedure, **ReadChar** will not print the character to the screen.

```
? readchar
Z           ("z" was typed)
? if equal? readchar "Y [print "yes]
```

ReadList will return an entire list from the keyboard. It prints the list as it is being typed, and it will let the user edit the list until the RETURN key has been pressed. ReadList will properly handle words, numbers, and lists.

```
? readlist
[kathy stan] [tammy pete] [donna jon] (RETURN)
[KATHY STAN] [TAMMY PETE] [DONNA JON]

? last readlist
37 74 [hup hup] hike (RETURN)
HIKE
```

Since both of these primitives will wait for a user's input your program should prompt the user for input.

Did you notice that the **ReadChar** waited for a single character to be typed but the **ReadList** accepted as many as you wanted and waited for a RETURN to signal the end?

Mouse returns the current X and Y coordinates of the mouse.

```
? mouse
[0 0]          (if the mouse is at home)
```

Button? returns TRUE if the mouse's left button is pressed.

```
? button? 0
false

? repeat 200 [print button? 0]
FALSE
FALSE
FALSE
TRUE            (mouse button pressed)
TRUE
TRUE
FALSE           (mouse button released)
```

If you take a minute, you can probably think of some game that could use these primitives.

Mastering Logo

This section will help you understand a few of the more advanced features of Logo.

Workspace Management

The *workspace* is where Logo keeps the names, definitions, and values for your procedures and variables. The workspace is a part of the computer's memory, and *it exists only while you are running Logo.* When you exit Logo or turn off your computer, everything in the workspace is lost. If you want to keep your workspace for later, you must save it on disk as a file (see *Load and Save* section in the *Using Logo* chapter).

Packages

When you start to write larger programs, you will fill your workspace with many procedures and variables. There will be times when you will want to gather together several procedures and variables into a single *package* that can be treated like a unit. A package can by hidden from workspace commands like **PrintOutAll**, **Save**, **EraseAll**, **EditAll**, etc. A package can be saved to disk, loaded from disk, edited, and printed to the screen. A package is like having a separate, independent workspace.

Suppose you have several procedures that you would like to keep together in a graphics package. You could create a package called **"graphics** and add the procedures to it:

```
? package ''graphics ''draw.box
? package ''graphics ''draw.arc
? package ''graphics ''draw.circle
```

Or you could add them all at once with a list:

```
? package "graphics [draw.box draw.arc draw.circle]
```

You could also make the entire workspace become the graphics package:

```
? pkgall "graphics
```

Once you've done this, you can perform several operations on this graphics package as a group: print, bury, save, edit, and erase. All you have to do is specify the package name along with the command. So

```
? poall "graphics
```

will print the entire contents of the graphics package, and

```
? save "graphics "graphics
```

will save the package to a file called GRAPHICS.

To hide the contents of a package so that it won't be affected by most of the other workspace commands, use **Bury**:

```
? bury "graphics
```

You can now forget about all the graphics procedures and variables, and you won't see them, edit them, or accidentally erase them, unless you want to.

To expose a package that has been buried, just **Unbury** it:

```
? unbury "graphics
```

This will make the graphics package visible in the workspace.

Most of the workspace primitives deal with packages:

Bury	Save	Load
EditAll	EditNames	EditProcs
EraseAll	EraseNames	EraseProcs
PrintOutAll	PrintOutNames	PrintOutProcs
PrintOutTitles		

See the primitives reference chapter for more details about specific commands.

Printing Out

Logo provides a number of commands for printing out the procedures and variables stored in your workspace.

PrintOut (PO) prints the name and definition for a procedure, and the name and value for a variable. It accepts the name of a procedure or variable as input, but it also accepts a list of procedures and variable names. Supplying no input will result in an error message.

All procedures and variables are printed in a format that can be directly entered into Logo. Procedures will include both the **To** and **End** words. Variables will be preceded with the **Make** word.

PrintOutAll (POAll) is similar to **PrintOut** but handles an entire package containing many definitions and variables. It accepts the name of a package or a list of packages as input. Supplying no input will default to printing all unburied procedures and variables in the workspace.

PrintOutNames (PONs) is similar to **PrintOutAll** but handles just the variables within a package. It accepts the name of a package or a list of packages as input. Supplying no input will default to printing all unburied variables in the workspace.

PrintOutProcs (POPs) is similar to **PrintOutAll** but handles just the procedures within a package. It accepts the name of a package or a list of packages as input. Supplying no input will default to printing all unburied procedures in the workspace.

PrintOutTitles (POTs) is similar to **PrintOutProcs** but prints just the titles lines (the first line of a **To** definition) of procedures within a package. **PrintOutTitles** accepts the name of a package or a list of packages as input. Supplying no input will default to printing all procedure titles in the workspace.

Erasing Names

If you are using Logo for a long period of time, it may become cluttered with a lot of old procedures and variables. These can be easily erased with a few workspace commands.

Erase removes the definition of a procedure or value of a variable. It accepts the name of a procedure or variable as input. It will also accept a list of procedures and variable names.

```
? erase "box
```

```
? erase [box square circle]
```

EraseAll (ErAll) is similar to **Erase** but handles an entire package containing many definitions and variables. It accepts the name of a package or a list of packages as input. Supplying no input will default to erasing all unburied procedures and variables in the workspace.

EraseNames (ErNs) is similar to **EraseAll** but handles just the variables within a package. It accepts the name of a package or a list of packages as input. Supplying no input will default to erasing all unburied variables in the workspace.

EraseProcs (ErPs) is similar to **EraseAll** but handles just the procedures within a package. It accepts the name of a package or a list of packages as input. Supplying no input will default to erasing all unburied procedures in the workspace.

In Logo new procedure definitions are usually created with the special word **To**. When a procedure is defined with **To** from the command window, the Logo screen editor is invoked. The editor will place you into the edit window, insert the procedure title line, and an **End**. You can then fill out the body of the procedure as you wish.

The edit window accepts all of the same editing keys as the command window. You are free to move and edit anywhere in the edit window. The big difference between the command and edit windows is in the effect of the RETURN key. From the command window a RETURN informs Logo to execute your line. In the edit window RETURN creates a new blank line.

When you have finished typing a procedure definition, press CTRL-C to exit the editor. Logo will read what you have typed and save it. If you've edited procedures, the procedures will be redefined.

If you want to exit, but don't want to save your changes press CTRL-G. The changes that you have made while in the editor will remain in the editor's buffer. The difference is that Logo will not interpret the changes and will not update the workspace. You can return to the editor buffer as modified by typing **Edit** without any inputs.

To edit an existing procedure, use the **Edit** command, followed by the procedure name (remember the leading quote):

    ? edit ''box

To resume an existing Logo editor session you need not supply the name of the procedure:

    ? edit

This is handy for switching back and forth between the command and editor windows.

Edit will also accept a list of procedure and variable names as input:

```
? edit [box square circle]
```

You can edit more than just procedures and variables in the editor. If you type one or more command lines that are not part of a procedure definition, they will be saved when you press CTRL-C. A word of caution, however: do not attempt to execute editor commands from within the editor itself. Logo would not understand.

Mouse Pointing

The mouse can be used to position the text cursor anywhere within the command or editor windows. This is a handy and quick way of moving the cursor position. Simply move the mouse pointer to the desired position, then click the left mouse button. The cursor will jump to its new location.

To access text off the top or bottom of the screen, you can use the normal scrolling keys or hold the left mouse button down and move the mouse to the top or bottom of the screen.

Normal Keys

This section describes a few of the normal keys used within the editor and command windows. They are all available on the standard keyboard, and do not require the CTRL key be held down.

Backspace Move cursor back, deleting the previous character.

When using BACKSPACE at the toplevel, the cursor will not back-up over the prompt.

Characters deleted with BACKSPACE cannot be recovered with CTRL-Y.

If the line wraps off the right edge of the screen, the cursor will move to the wrap point on the previous line.

This key will not delete back further than the beginning of a line. If you want to join two lines, use CTRL-K.

CTRL-H performs the same operation as BACKSPACE.

Del   Delete the character under the cursor.

This key does not move the cursor back like BACKSPACE.

If the cursor is at the end of a line, this key *will not* cause the next line to be joined to the current line. If you want to join two lines, use CTRL-K.

Characters deleted with DEL cannot be recovered with CTRL-Y.

CTRL-D performs the same operation as DEL.
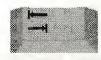
Return   Enter a line and begin a new line.

If RETURN is pressed in the Command window, the line is evaluated by Logo. In the Edit window, a new line is created.

The RETURN key will not split lines, nor do you need to have the cursor at the end of a line in order to enter the entire line.

Tab   Tab forward to the next tab stop.

Logo will fill with spaces to the next tab stop. BACKSPACE will move the cursor back a single space, not to the previous position.

Tab stops are set to every four positions.

Up-Arrow   Move cursor to the previous line.

If there are lines off the top of the screen, when the cursor reaches the top, the screen will scroll down.

Down-Arrow	Move cursor to the next line.
	If there are lines off the bottom of the screen, when the cursor reaches the bottom, the screen will scroll up.
Left-Arrow	Move back one character.
	Move over the character, do not delete it. If the left edge of the screen is reached, the cursor will wrap to the previous line.
Right-Arrow	Move forward one character.
	If the right edge of the screen is reached, the cursor will wrap to the next line.
Shift-Up-Arrow	Move back a half page.
	If there are lines off the top of the screen, scroll the page down.
Shift-Down-Arrow	Move forward a half page.
	If there are lines off the bottom of the screen, scroll the page up.
Shift-Left-Arrow	Move to the beginning of a line.
	This moves the cursor to the beginning of the text line, even if the line wraps across the edge of the screen.
Shift-Right-Arrow	Move to the end of a line.
	This moves the cursor to the end of the text line, even if the line wraps across the edge of the screen. If the cursor is past the end of the line, this key will move the cursor back to the end of the text line (ignoring spaces).
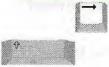
Control Keys

These keys combine the CTRL key and another key. Press CTRL and hold it while you type the other key. Then you can release the CTRL key.

Ctrl-A   Move to the beginning of a line.

This moves the cursor to the beginning of the text line, even if the line wraps across the edge of the screen.

Ctrl-B   Move back one character.

Move over the character, do not delete it. If the left edge of the screen is reached, the cursor will wrap to the previous line.

Ctrl-C   Exit the editor and evaluate all changes.

Upon exiting, all lines in the editor buffer are re-entered into Logo, just as if the lines were typed from the Command window.

Ctrl-D   Delete a character.

The character under the cursor is deleted. This key does not move the cursor back like BACKSPACE or Ctrl-H.

If the cursor is at the end of a line, this key *will not* cause the next line to be joined to the current line. If you want to join two lines, use CTRL-K.

Characters deleted with DEL cannot be recovered with CTRL-Y.

CTRL-D performs the same operation as DEL.

Ctrl-E   Move to the end of a line.

This moves the cursor to the end of the text line, even if the line wraps across the edge of the screen. If the

cursor is past the end of the line, this key will move the cursor back to the end of the text line (ignoring spaces).

Ctrl-F Move forward one character.

If the right edge of the screen is reached, the cursor will wrap to the next line.

Ctrl-G Cancel operation.

When in the Edit window, exit the editor, and do not evaluate changes.

When in the Command window, stop the current operation.

Ctrl-H Move cursor back, deleting the previous character.

When using CTRL-H at the toplevel, the cursor will not back-up over the prompt.

Characters deleted with CTRL-H cannot be recovered with CTRL-Y.

If the line wraps off the right edge of the screen, the cursor will move to the wrap point on the previous line.

This key will not delete back further than the beginning of a line. If you want to join two lines, use CTRL-K.

Ctrl-I Tab forward to the next tab stop.

Logo will fill with spaces to the next tab stop. BACKSPACE will move the cursor back a single space, not to the previous position.

Tab stops are set to every four positions.

Ctrl-J   Start a new line.

Terminate the current line and start a new line. If the cursor is inside the line, the line will be split.

Ctrl-K   Kill to end of line.

Kill (delete) all text from the cursor to the end of the line. This text is placed in the kill buffer, and may be replaced or inserted elsewhere with Ctrl-Y.

If CTRL-K is pressed when the cursor is at the end of a line, the next line will be joined to the end of the current line. This is as if the CTRL-K deleted the line separation character.

To delete an entire line of text, press CTRL-A, CTRL-K, CTRL-K.

Ctrl-L   Center a line.

Scroll the screen up or down until the current line is in the center.

Ctrl-M   Enter a line and begin a new line.

If CTRL-M is pressed in the Command window, the line is evaluated by Logo. In the Edit window, a new line is created.

The CTRL-M key will not split lines, nor do you need to have the cursor at the end of a line in order to enter the entire line.

Ctrl-N   Move cursor to the next line.

If there are lines off the bottom of the screen, when the cursor reaches the bottom, the screen will scroll up.

Ctrl-O   Open a line.

A new line is created at the current cursor position.

If this is done inside an existing line, the line will be split into two parts.

Ctrl-P   Move cursor to the previous line.

If there are lines off the top of the screen, when the cursor reaches the top, the screen will scroll down.

Ctrl-Q   Move back a word.

Ctrl-R   Move back a half page.

If there are lines off the top of the screen, scroll the page down.

Ctrl-S   *Reserved*. Does nothing.

Ctrl-T   Toggle between text and graphics windows.

Ctrl-V   Move forward a half page.

If there are lines off the bottom of the screen, scroll the page up.

Ctrl-W   Forward a word.

Ctrl-Y   Yank back text.

Insert the text deleted with Ctrl-K.

This is how lines are moved and copied in the editor.

Ctrl-Z   Pause. See the explanation for **Pause**.

EditAll is similar to **Edit** but handles an entire package containing many definitions and variables. It accepts the name of a package or a list of packages as input. Supplying no input will default to editing all procedures and variables in the workspace.

EditNames is similar to **EditAll** but edits only the variables within a package. It accepts the name of a package or a list of packages as input. Supplying no input will default to editing all variables in the workspace.

EditProcs is similar to **EditNames** but edits only the procedures within a package.

Editing Files

EditFile lets you edit a Logo file or a simple text file directly. It accepts the name of a Logo disk file as input. For example:

```
? editfile "init
```

Will edit the file INIT.

When you exit with CTRL-C the contents of the editor will be executed, *but the file will not be written back to disk*. To save your changes back to disk, use **SaveEdit (SaveFile)**:

```
? savefile "init
```

If you exit with CTRL-G the contents of the editor will not be executed. Any procedures will be left unchanged.

Suppose you want to run a procedure called **box** that is stored with a file named *BOX*. The *BOX* file already contains the text (as the result of a **Save**):

```
to box
repeat 4 [forward 50 right 90]
end
```

then you would type:

```
? editfile "box
```

which would enter the editor. Add the word **box** to the end of the file:

```
to box
repeat 4 [forward 50 right 90]
end

box
```

Exit the editor with CTRL-G (so as not to execute the editor buffer causing **box** to run right now), and type:

```
? savefile "box
```

to save your change. The next time you load the *box* file, Logo will run the **box** procedure automatically. From the Amiga's Workbench, if you click on the *BOX* icon, Amiga Logo will load the **box** procedure and run it.

Do not attempt to execute **EditFile** from within the editor itself. Logo will not allow it.

If you no longer require a particular file, you can use the EraseFile command to erase it permanently from disk:

```
? erasefile "junk
```

Use this command with care. Erased files cannot be recovered.

Printing Hardcopy

Depending on whether you use Workbench 1.2 or 1.3, there are different methods for printing with Amiga Logo.

If you have Workbench 1.2:

Open up the CLI by double clicking on the CLI icon in the System drawer. Copy the printer driver(s) of

your choice from your Workbench disk to the devs/
printers directory of your Logo disk. Go into
Preferences on the Logo disk and change your printer
to the one desired and save the setting. Now when
you boot with the Amiga Logo disk, the correct
printer driver will be installed.

If you have Workbench 1.3:

Run the InstallPrinter program to copy printer drivers
from the Extras 1.3 disk to the Amiga Logo disk.
Select the correct printer driver in the Change Printer
window of Preferences, and save the setting.

If you boot with Workbench 1.2 or 1.3, and then load
Amiga Logo, Logo will use the printer driver installed
on the Workbench disk.

Amiga Logo supplies commands for printing both
text and graphics to your printer.

The **DumpText** (**DT**) command will print the entire
contents of your text window (including lines off the
top and bottom) to the printer.

```
? cleartext pots
. . .
? dumptext
```

The **DumpEdit** (**DE**) command will print the entire
contents of your editor buffer to the printer. For
example, let's say you want to print the contents of
the INIT file:

```
? editfile "init
<press ctrl-G when the editor window comes up>
? dumpedit
```

The **DumpGraphics** (**DG**) command will print the
graphics window to the printer. The quality of the
output will depend on your printer and your choices
of colors. Use the Amiga preferences program to
select a printer and set its graphics printing options.
Note that you will need to install printer drivers from
the AmigaDos Extras Disk to the Amiga Logo system
disk.

Definition Lists

There is an alternate way of defining procedures that does not require the editor. The **Define** primitive will create a new procedure from two inputs, a name and a definition list. This command can be performed within another procedure, so procedures can create other procedures.

Here is an example of using **Define**:

```
? define "box [ [size] [repeat 4 [rt 90 fd :size] ]
```

The first input is the new procedure name. The second input is a procedure definition list. Its first element is a list of input names to the procedure. These names do not need a ':' before them. If there are no inputs, a [ ] should be used. The rest of the definition list contains the executable lines of the procedure definition. Each line is itself a list.

Define lets you piece together your procedure under the control of Logo:

```
? make "def [ [size color] [fd :size] ]
? make "def lastput [setpc :color] :def
? define "line :def
```

The **Text** operation will output the definition for an existing procedure.

```
? text "line
[ [SIZE COLOR] [FD :SIZE] [SETPC :COLOR] ]
```

This gives you another source of valid input to **Define**.

```
? define "line90 lastput [rt 90] text "line
```

What power!

CopyDef is a simple way to make a copy of a procedure definition. The new copy is given a name with which it can be executed or edited just like any other procedure.

For example you could copy the procedure defined previously:

```
? copydef "square "box
```

Now you have an identical procedure named **square**:

```
? text "square
[ [SIZE] [REPEAT 4 [RT 90 FD :SIZE] ] ]
```

If you edit the procedure for **square** the **box** procedure will not change.

Procedures as Variables

Amiga Logo tries not to distinguish between names for variables and names for procedures. Unlike other Logos, the same name cannot be both a value and a procedure at the same time. Within a procedure definition a local variable can have the same name as a procedure, but you would not be able to call that procedure.

In Amiga Logo, you can make a variable with a value that is a procedure—*not the output of a procedure—an actual procedure.* If you type:

```
? make "bx :box
```

you are asking Logo to create a variable named **bx** with the value of the variable **box**. With the colon (:),

you are asking for a value, not an execution. Since box is a procedure, **bx** is now a procedure as well:

```
? text "bx
[ [SIZE] [REPEAT 4 [RT 90 FD :SIZE] ]

? bx 30
```

Although it is less useful, this also works for primitives:

```
? make "fwd :forward

? fwd 40
```

So, you can now give your own names to primitives.

Reference

As you are learning to program with Logo, you will need to refer to this section for information about the function of various primitives, required and optional inputs, correct syntax, the meaning of error messages, etc. While other books about Logo will serve well in teaching you the basics of Logo programming, this section is your complete reference guide to Amiga Logo.

The Introduction (page 104) describes the basic parts of a Logo primitive definition.

On pages 105 and 106 you will find a definition list of all inputs used with the Logo primitives.

Pages 106-195 contain detailed descriptions and examples of each Logo primitive arranged in alphabetical order, beginning with mathematical operators.
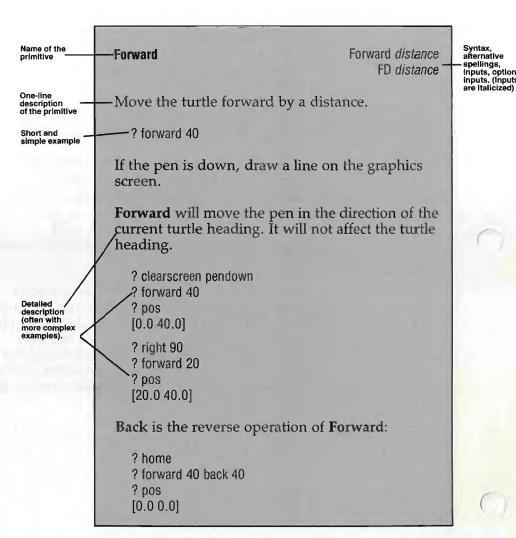
A summary of error messages is provided on pages 196-202.

Included with the Amiga Logo package is a quick reference card which gives short definitions of Logo primitives. The card has been designed to allow you easy access to a reference source. Since you may sometimes need a more detailed description about a certain primitive, each entry on the reference card has been cross referenced to the appropriate entry in this reference section.

This reference section describes each of the Logo inputs and primitives in alphabetical order.

The following example shows the different parts of a primitive description:

Name of the primitive —— **Forward**   Forward *distance*   **Syntax, alternative spellings, inputs, options. (Inputs are Italicized)**

FD *distance*

One-line description of the primitive —— Move the turtle forward by a distance.

Short and simple example —— ? forward 40

If the pen is down, draw a line on the graphics screen.

Forward will move the pen in the direction of the current turtle heading. It will not affect the turtle heading.

```
? clearscreen pendown
? forward 40
? pos
[0.0 40.0]

? right 90
? forward 20
? pos
[20.0 40.0]
```

Detailed description (often with more complex examples).

Back is the reverse operation of **Forward**:

```
? home
? forward 40 back 40
? pos
[0.0 0.0]
```

As mentioned above, the inputs to a primitive are shown to the right. An *italicized* word is used to represent the type of input required. The possible types are listed below:

angle	An angle in degrees.
character-word	A single character as a word.
color-number	A number representing a color.
column	Horizontal position of a character in the text window.
directory	An AmigaDOS directory name or path.
distance	A number representing a distance on the graphics screen.
file	A disk file name.
input	The input to a procedure definition.
label	A word representing a position in a procedure.
list	A set of items enclosed as a unit.
name	A word representing a variable or procedure.
name-list	A list of names.
number	A decimal or integer number.
object	A word, number, or list.
package	A word representing a group of variables and procedures.
package-list	A list of packages.

pen-state	The state of the pen: up, down, erase, or reverse.
period	Time in 1/60ths of a second.
pred	A TRUE or FALSE value.
property	A word representing a property of a name.
row	The vertical position of a character in the text window.
run-list	A list containing objects that can be executed by Logo.
word	A sequence of characters.
x	The horizontal position on the graphics screen.
y	The vertical position on the graphics screen.
. . .	Indicates more of the same input type.

Primitives

+
$$number + number$$
$$+ number number$$
$$(+ number number . . .)$$

sum *number number*
(sum *number number* . . .)

Output the sum of the input numbers.

```
? 2 + 5
7

? 2 + 5 + 7 + 10
24
```

Add the inputs together and output their sum.

This operator can be used in a prefix fashion, where both inputs follow the operator. This format is perhaps a little more difficult to read, but is consistent with how all other Logo functions are used.

```
? + 2 5
7
```

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

```
? (+ 2 5 7 10)
24
```

—   *- number*
                                              number - number
                                               - number number
                                        (*- number number . . .*)

                                    difference *number number*
                              (difference *number number . . .*)

Output the difference of the input numbers.

```
? 5 - 3
2
? 5 - 3 - 4
-2.0
```

Subtract one input from another and output their difference.

Notice that there must be a space between the minus (-) and its second input. If this were not done the second input would be read as a negative number, and there would be no operator, so an error might result.

```
? 5 -3
5
I don't know what to do with -3
```

This primitive can also be used to negate a number.

```
? - 10
-10
```

This operator can be used in a prefix fashion, where both inputs follow the operator. This format is perhaps a little more difficult to read, but is consistent with how all other Logo functions are used.

```
? - 5 2
3
```

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

```
? (- 5 3 4)
-2.0
```

*

> *number * number*
> ** number number*
> (** number number* . . .)
>
> product *number number*
> (product *number number* . . .)

Output the product of the input numbers.

```
? 2 * 3
6.0

? 2 * 3 * 10
60.0
```

Multiply the inputs and output their product.

This operator can be used in a prefix fashion, where both inputs follow the operator. This format is perhaps a little more difficult to read, but is consistent with how all other Logo functions are used.

```
? * 2 3
6.0
```

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

```
? ( * 2 3 10 )
60.0
```

/

number / number
/ number number
(/ number number . . .)

quotient number number
(quotient number number . . .)

Output the quotient of the input numbers.

```
? 10 / 2
5.0

? 10 / 2 / 5
1.0
```

Divide the inputs and output their quotient.

This operator can be used in a prefix fashion, where both inputs follow the operator. This format is perhaps a little more difficult to read, but is consistent with how all other Logo functions are used.

```
? / 10 2
5.0
```

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

```
? ( / 10 2 5 )
1.0
```

<

number < number
< number number

Output TRUE if first number is less than second number.

```
? 2 < 3
TRUE

? 2 < 1
FALSE
```

This operator can be used in a prefix fashion, where both inputs follow the operator. This format is perhaps a little more difficult to read, but is consistent with how all other Logo functions are used.

```
? < 2 3
TRUE
```

>

number > number
> number number

Output TRUE if first number is greater than second number.

```
? 2 > 1
TRUE

? 2 > 3
FALSE
```

This operator can be used in a prefix fashion, where both inputs follow the operator. This format is perhaps a little more difficult to read, but is consistent with how all other Logo functions are used.

```
? > 2 1
TRUE
```

object = *object*
= *object object*
Equal? *object object*

Output TRUE if an object equals another.

```
? 2 = 2
TRUE

? "apple = "peach
FALSE

? [grape banana] = [grape banana]
TRUE
```

This operator works for numbers, lists, and words. Numbers are equal if they are numerically the same. Words are equal if they have the same characters in identical order. Lists are equal if their elements are each equal.

This operator can be used in a prefix fashion, where both inputs follow the operator. This format is perhaps a little more difficult to read, but is consistent with how all other Logo functions are used.

```
? = 12 12
TRUE
```

The **Equal?** primitive performs the same operation.

Abs

Abs *number*

Output the absolute value of a number.

```
? abs -10
10

? abs 10
10

? abs -3.1
3.1
```

Notice that regardless of whether you input a
negative or positive number, the output will always
be a positive number.

Here is a graphics example where this might be
useful:

```
? home left 90 forward 25
? pos
[-25.0 0]
? abs first pos
25.0
```

And *pred pred*
(And *pred pred* . . .)

Output TRUE if all inputs are TRUE.

```
? and ''true ''true
TRUE
```

```
? and ''true ''false
FALSE
```

The inputs to **And** are called *predicates*. They must be
either TRUE or FALSE.

For example, the comparison of two numbers outputs
either TRUE or FALSE, so this could be used as input
to **And**:

```
? and (1 < 10) (40 > 20)
TRUE
```

If you attempt to input something other than TRUE or FALSE, Logo will inform you that you made a mistake:

? and 3 4
3 is not true or false

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

? (and "true "true "false "true)
FALSE

Here is an example of how **And** might be used in graphics:

? setpos [45 72]
? make "x first pos
? make "y last pos
? (and (:x > 0) (:x < 160) (:y > 0) (:y < 100))
TRUE

ArcTangent   ArcTangent *number*
                                                      ArcTan *number*

Output the angle whose tangent is the input number.

? arctan 1.0
45.0

This function does the *inverse* operation of **Tangent**: it outputs the number of degrees whose tangent is the input number. This is called the arc-tangent.

? arctan tan 80
80.0

Output the ASCII number representing a character.

```
? ascii ''a
65

? ascii ''?
63
```

ASCII only outputs the number for the first letter of the input. If a longer input is used, all other letters of the input word will be ignored.

```
? ascii ''starfish
83

? ascii 123
49
```

Note: Logo shifts lowercase characters to uppercase during input. This means that inputs of 'a' and 'A' will produce the same output.

ASCII performs the reverse operation of **Char** (see the description below):

```
? ascii char 40
40

? char ascii ''a
A
```

Aspect   Aspect
                                                                   Scrunch

Output the aspect ratio of the screen.

```
? aspect
1.0
```

Aspect returns the current aspect ratio of the screen. This ratio controls the scale of the Y axis compared with the X axis. It is the number of units in the Y direction for each unit in the X direction.

The screen aspect ratio can be changed with **SetAspect**.

SetAspect and **Aspect** are provided to compensate for different screen resolutions and monitor brands. If your circles look like ellipses and squares look like rectangles, the aspect ratio can correct these.

For example, to make each vertical unit half the size of a horizontal unit, an aspect ratio of 0.5 would be used.

Scrunch is the old word for **Aspect**.

Back

Back *distance*
BK *distance*

Move the turtle backward a distance.

```
? back 40
```

If the pen is down, draw a line on the graphics screen.

Back will move the pen in the direction opposite the current turtle heading. It will not affect the turtle heading.

```
? clearscreen pendown
? back 40
? pos
[0 -40.0]
? right 90
? back 20
? pos
[-20.0 -40.0]
```

Back is the reverse operation of **Forward** (described below):

```
? home
? forward 40 back 40
? pos
[0 0]
```

Output the color number of the screen.

```
? background
0
```

The color number output is an index into the screen color table. This number can be set with **SetBackground**.

The actual background color displayed depends on the color setting for this color number. For more information see the explanation for **RGB**.

For example:

```
? setpc background
```

would set the pen color to that of the background. This would, in effect, make the pen erase.

Hide a package from various workspace commands.

? bury "my.graphics

All procedures and variables defined within the specified package will be "hidden" from certain workspace commands. This is a good way to hide procedures and variables that you have finished. Often you will want to save utility procedures for reuse in several different programs, and you don't want all these utility procedures to get in the way of the program you are developing.

The following workspace primitives are affected by **Bury**:

Save	PrintOutTitles	
EditAll	EditNames	EditProcs
EraseAll	EraseNames	EraseProcs
PrintOutAll	PrintOutNames	PrintOutProcs

A package will remain buried until an **Unbury** command is given. For additional information, see **Package**.

ButFirst   ButFirst *object*
                                                                                                BF *object*

Output all but the first element of an object.

? butfirst [book pen ruler]
[PEN RULER]

```
? butfirst "fish
ISH

? butfirst 1234
234

? butfirst butfirst [red green blue]
[blue]
```

This operation works for both lists and words. If the input is a list, the output of **ButFirst** will be the same list, without the first element. If the input is a word, all but the first character are output. Keep in mind that numbers are words as well.

It is an error for the input object to be empty:

```
? butfirst [   ]
BUTFIRST does not like [   ] as input
```

ButLast   ButLast *object*
                                                                BL *object*

Output all but the last element of an object.

```
? butlast [book pen ruler]
[BOOK PEN]

? butlast "fish
FIS

? butlast 1234
123

? butlast butlast [red green blue]
[red]
```

This operation works for both lists and words. If the input is a list, the output of **ButLast** will be the same list, without the last element. If the input is a word, all but the last character are output. Keep in mind that numbers are words as well.

It is an error for the input object to be empty:

? butlast [   ]
BUTLAST does not like [   ] as input

Button?

<div align="right">

Button? *number*
ButtonP *number*

</div>

Output TRUE if the mouse's left button is pressed.

? button? 0
FALSE

Button? will output TRUE as long as the mouse button is held down. When the button is released, it will output a FALSE.

The number input indicates what mouse input port to check. A zero indicates the primary mouse port. A one is for the secondary mouse port.

? repeat 300 [print button? 0]
FALSE
FALSE
FALSE
TRUE (button pressed)
. . .

ButtonP is the old way of spelling **Button?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Catalog

<div align="right">

Catalog
Catalog *directory*

</div>

Print the names of all files in a directory.

```
? catalog
DEMO
DEMO.INFO
INIT
INIT.INFO
AMIGALOGO
AMIGALOGO.INFO
. . .
```

Catalog takes an optional input to specify the disk file
directory to print. If an input is not specified, the
current directory will be printed. When specified the
input may include a disk name and directory path.
For example:

```
? catalog "df0:devs
```

would print the names of files in the DEVS directory
of floppy disk DF0.

Catch   Catch *name run-list*

Trap an error or the result of a **Throw**.

Catch is used to handle exceptions to the normal flow
of control in your program. Exceptions are generated
as the result of an error, or they can be generated by
using the **Throw** command.

The first input is a name used to identify this **Catch** to
its corresponding **Throw**. The **Throw** command
should specify the same name.

The second input is a list to execute. If it encounters a
throw during its execution, control returns from this
command.

With the **Catch** command errors can be trapped by a
program before they are printed and processed by
Logo. This is done by using the word "**Error** as the
catch label. When an error occurs, control will be
transferred to the most recent catch of this type. See
Error for more information.

Output the letter which is the ASCII character for a number.

```
? char 65
A

? char 52
4
```

The input must be a number between 0 and 255. The output will be a word which contains the corresponding ASCII character.

Char performs the reverse of the **ASCII** operation.

```
? char ascii "X
X
```

Clear the graphics screen without affecting the turtle.

```
? setpos [10 10]
? clean
? pos
[10.0 10.0]
```

This command erases everything on the graphics screen, but does not affect the turtle's position, heading, color, or penstate.

The commands:

```
? home
? clean
? pendown
```

would produce the same result as **ClearScreen**.

ClearScreen

ClearScreen
CS

Clear the graphics screen and home the turtle.

? clearscreen

The entire graphics screen will be erased, and the turtle will be positioned at its home position (the center of the screen). Its turtle heading will be set to zero (north) and its pen will be put down.

ClearText

ClearText
CT

Clear the text window, and home the text cursor.

? cleartext

Everything in the text window will be erased and the text cursor will be positioned to line one.

All text including that which is off the top or the bottom of the text window will be erased. This is a handy command to keep the screen from becoming too cluttered.

When a **ClearText** is executed, all of the memory used by text is freed for reuse in Logo.

Continue

Continue
Co

Resume execution after a pause.

? continue

Both the **Pause** command and CTRL-Z let you stop the execution of a procedure. **Continue** will resume execution from where the pause occurred.

Copy a procedure definition to a new name.

```
?copydef "newstar "oldstar
```

CopyDef is a simple way to make a copy of a procedure definition. The copy is given a new name with which it can be executed or edited just like any other procedure.

Suppose you had a procedure named "box:

```
? define "box [[ ] [repeat 4 [rt 90 fd 20]]]
```

You could copy it:

```
? copydef "square "box
```

and now you have an identical procedure named "square:

```
? square
? text "square
[[ ] [REPEAT 4 [RT 90 FD 20]]]
```

If you edit the procedure for "square, the "box procedure will remain unchanged.

Another way to make a copy of a procedure is with the **Make** primitive.

Cosine   Cosine *angle*
                                                              Cos *angle*

Output the cosine of an angle.

```
? cos 60
0.5
```

Given an input number representing an angle in degrees, **Cosine** will output its cosine value. The cosine of an angle will always be in the range of 1 to 0 to -1.

```
? cosine 0
1.0

? cosine 90
0

? cosine 180
-1.0
```

The cosine of an angle is very useful for many types of graphics operations. For example:

```
? clearscreen
? right 75 forward 80
? home
? forward (80 * cosine 75)
```

Both lines will be drawn with the same distance in the Y direction.

Count   Count *object*

Output the number of elements in an object.

```
? count [apple orange grape]
3

? count ''grape
5

? count 1234
4

? count []
0
```

Count works for both lists and words. If the input is a list, the number of items in the list is output. If the input is a word, the number of characters in the word is output. Keep in mind that numbers are words as well.

Cursor   Cursor

Output the position of the text cursor.

```
? cleartext
? cursor
[0 1]
```

Cursor outputs a list containing the row and column positions of the cursor within the text window.

The upper left cursor position of the text window is [0 0]. The maximum values for the row and column of the text cursor will depend on the size of the text window.

The text cursor can be positioned with **SetCursor**.

Define   Define *name list*

Make a definition list into a procedure.

```
? define "box [[size] [repeat 4 [rt 90 fd :size]]
? box 50
```

Define will create a procedure from a list. This command can be executed within another procedure or at the top level the Logo editor is not required.

The first input supplied to **Define** is the new procedure name.

The second input is a list which specifies the procedure. The first element is a list of input names to the procedure. These names do not need a ':' before them. If there are no inputs, a [ ] should be used.

The rest of the list is the executable part of the procedure definition. Each procedure line is itself a list.

```
? define "south [ [] [setheading 180]]

? define "segs [[a size] [repeat 8 [rt :a fd :size]] ]
```

The **Text** command can be used to output a procedure definition.

Define?

Output TRUE if the input object is a procedure.

```
? define? "box
TRUE

? define? "forward
FALSE

? define? 12
FALSE
```

DefineP is the old way of spelling **Define?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Difference

difference *number number*
(difference *number number...*)

- *number*
number - *number*
- *number number*
(- *number number...*)

Output the difference of the input numbers.

See the description for "-".

Output the names of all files in a directory.

```
? dir
[DEMO DEMO.INFO INIT INIT.INFO AMIGALOGO . . . ]
```

Dir takes an optional input to specify the disk file directory to print. If an input is not specified, the current directory will be printed. When specified the input may include a disk name and directory path. For example:

```
? dir "df0:devs
```

would print the names of files in the DEVS directory of floppy disk DF0.

Due to the 240 character limit, DIR cannot list all of a long index or directory; it will stop when it has reached its limit.

Dot   Dot *[x y]*

Put a dot at an X,Y position without affecting the turtle.

```
? dot [-50 50]
```

Dot expects a position list as input. The first element of the list is the x position, the second is the y position.

As in other drawing commands, the dot will be drawn in the color of the graphics pen. **Dot** is not affected by other pen settings.

Dump the contents of the edit buffer to the printer.

? dumpedit
DUMPING TO PRINTER

The **DumpEdit (DE)** command will print the entire
contents of your editor buffer to the printer. So, for
example, you could print the contents of the INIT file
with:

? editfile ''init
<press ctrl-G when the editor window comes up>
? dumpedit

Dump the graphics screen to the printer.

? dumpgraphics
DUMPING TO PRINTER

The **DumpGraphics (DG)** command will print the
graphics window to the printer. The quality of the
output will depend on your printer and your choices
of colors. Use the Amiga preferences program to
select a printer and set its graphics printing options.

Dump the entire text window to the printer.

? dumptext
DUMPING TO PRINTER

The **DumpText (DT)** command will print the entire
contents of your text window (including lines off the
top and bottom) to the printer.

? cleartext pots

. . .
? dumptext

Edit
Edit *name*
Edit *name-list*
Ed
Ed *name*
Ed *name-list*

Start or resume a Logo editor session.

```
? edit
```

This command is described in detail in the tutorial section of this manual.

Briefly:

Edit accepts the name of a procedure or variable as input. It will also accept a list of procedure and variable names as input. Supplying no input will return you to the previous edit session.

To exit the editor, type CTRL-C to save the contents of the editor buffer or CTRL-G to escape without saving.

For example:

```
? define "fish [ [] [make "goldfish "yellow]]
? edit "fish
```

Logo then switches to the editor window and prints:

```
TO FISH
MAKE "GOLDFISH "YELLOW
END
```

Do not attempt to execute **Edit** from within the editor itself. Logo will not allow it.

EditAll
EditAll *package*
EditAll *package-list*

EdAll
EdAll *package*
EdAll *package-list*

Edit everything in the workspace, a package, or packages.

? editall

EditAll is similar to **Edit** but handles an entire package containing many definitions and variables. **EditAll** accepts the name of a package or a list of packages as input. Supplying no input will default to editing all procedures and variables in the workspace.

To exit the editor, type CTRL-C to save the contents of the editor buffer or CTRL-G to escape without saving.

See the editor tutorial for more information.

Do not attempt to execute **EditAll** from within the editor itself. Logo will not allow it.

EditFile

Edit a Logo file.

? editfile "init

EditFile lets you edit a Logo file directly. It accepts the name of a Logo disk file as input.

To exit the editor, type CTRL-C to execute the contents of the editor buffer or CTRL-G to escape without executing.

EditFile does not automatically save your file changes back to disk. To save your changes back to a file you must use the **SaveEdit** (**SaveFile**) command.

See the editor tutorial for more information.

Do not attempt to execute **EditFile** from within the editor itself. Logo will not allow it.

EditNames

<div align="right">

EditNames
EditNames *package*
EditNames *package-list*

EdNs
EdNs *package*
EdNs *package-list*

</div>

Edit all variables in the workspace, a package, or packages.

? editnames

EditNames is similar to **Edit** but handles all variables within an entire package. **EditNames** accepts the name of a package or a list of packages as input. Supplying no input will default to editing all variables in the workspace.

To exit the editor, type CTRL-C to save the contents of the editor buffer or CTRL-G to escape without saving.

See the editor tutorial for more information.

Do not attempt to execute **EditNames** from within the editor itself. Logo will not allow it.

EditProcs
EditProcs *package*
EditProcs *package-list*

EdPs
EdPs *package*
EdPs *package-list*

Edit all procedures in the workspace, a package, or packages.

? editprocs

EditProcs is similar to **Edit** but handles all procedures within an entire package. **EditProcs** accepts the name of a package or a list of packages as input. Supplying no input will default to editing all procedures in the workspace.

To exit the editor, type CTRL-C to save the contents of the editor buffer or CTRL-G to escape without saving.

See the editor tutorial for more information.

Do not attempt to execute **EditProcs** from within the editor itself. Logo will not allow it.

Empty?

Empty? *object*
EmptyP *object*

Output TRUE if an object has no elements.

? empty? []
TRUE

? empty? [apple grape]
FALSE

? empty? "banana
FALSE

Empty? checks to see if the object has any elements. It works for both lists and words. If the input is a list, it

checks for list elements. If the input is a word, it checks for characters.

EmptyP is the old way of spelling **Empty?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

End

Terminate a procedure definition.

End tells Logo that there are no more expressions to execute in the current procedure. It must be placed at the end of every procedure.

End should not be used within the body of a procedure (see **Stop** and **Output**).

This command can only be used within a procedure.

Equal?

Equal? *object object*
EqualP *object object*

object = *object*
= *object object*

Output TRUE if the input objects are equal.

See the description for ' = ' above.

Erase

Erase *name*
Erase *name-list*

Er *name*
Er *name-list*

ErN *name*
ErN *name-list*

Erase a procedure or variable from the workspace.

Erase removes the definition of a procedure or value of a variable. It accepts the name of a procedure or variable as input. It will also accept a list of procedures and variable names.

EraseAll

Erase everything in the workspace, a package, or packages.

EraseAll is similar to **Erase** but handles an entire package containing many definitions and variables.

EraseAll accepts the name of a package or a list of packages as input. Supplying no input will default to erasing all unburied procedures and variables in the workspace.

EraseFile

Erase a file from the disk.

```
? save "work
? erasefile "work
```

EraseFile requests Logo to erase a disk file. Once erased a file cannot be recovered.

The file name can also include a device name and directory path. For example:

```
? erasefile "df0:logo\/test
```

If the file cannot be found, Amiga Logo will produce an error message:

```
? erasefile "works
File WORKS not found
```

EraseNames

EraseNames
EraseNames *package*
EraseNames *package-list*

ErNs
ErNs *package*
ErNs *package-list*

Erase all variables in the workspace, a package, or packages.

EraseNames is similar to **EraseAll** but handles just the variables within a package. **EraseNames** accepts the name of a package or a list of packages as input. Supplying no input will default to erasing all unburied variables in the workspace.

EraseProcs

EraseProcs
EraseProcs *package*
EraseProcs *package-list*

ErPs
ErPs *package*
ErPs *package-list*

Erase all procedures in the workspace, a package, or packages.

EraseProcs is similar to **EraseAll** but handles just the procedures within a package. **EraseProcs** accepts the name of a package or a list of packages as input. Supplying no input will default to erasing all unburied procedures in the workspace.

Error

Error

Outputs information about the last error.

Error outputs a list containing information about the most recent error that occurred. This list contains

- the error number;

- the name of the procedure in which the error occurred (or [] if the error did not occur in a procedure);

- the name of the primitive that caused the error;

- the input value possibly in question;

- the procedure line being executed (optional).

With the **Catch** command errors can be trapped by the program before they are printed and processed by Logo. See the **Catch** command.

Exit   Exit
                                                                              Quit

Exit from Logo and return to Workbench or CLI.

? exit

Exit will terminate this session of Logo and return you to the Workbench, the CLI, or whatever the program was that started Logo.

Prior to exiting Logo you may want to save your changes. See the **Save** and **SaveFile** commands below.

Fence   Fence

Restrict the turtle to moving only on the graphics screen.

? fence

Fence forces the edges of the screen to act as a boundary that stops the turtle. If you attempt to move the turtle beyond the edge of the screen, Logo will inform you:

? fence
? clearscreen
? left 30
? forward 1000
Turtle out of bounds

The fence can be removed with the **Window** and **Wrap** commands.

Fill an area bounded by the current pen color.

    ? fill

This command will flood-fill a portion of the screen with the current pen color. It fills in all directions an area that is already enclosed by lines of the current pen color.

The fill will begin at the current pen position, and *flood* outward until it hits a pixel on the screen that is the same color as the pen. If you change the pen color, you will end up with a different result. Also, the pen should be down for this command to work properly.

PenDown, **PenErase** and **PenReverse** apply to the **Fill** command similar to the pen.

```
? cs setpc 1
? repeat 4 [fd 80 rt 90]
? setpos [10 10]
? fill
```

If the area being filled is not closed, the fill may "leak-out" and cover the entire screen.

Fill an area bounded by a color.

    ? fillin

This command will flood-fill a portion of the screen with the current pen color. It fills an area that is already closed or filled in. The fill will start to the immediate right of the current pen position and spread out in all directions until it hits a line or area of a color different to that in which it started the fill.

The reason the fill starts to the immediate right of the pen position is to avoid filling a line just drawn by the

pen. If **Fillin** does not seem to work, try moving to the starting position with the pen up, then put the pen down immediately before the fill.

PenDown, **PenErase** and **PenReverse** apply to the **Fillin** command similar to the pen.

If the area being filled is not closed, the fill may "leak-out" and cover the entire screen.

```
? cs setpc 1
? repeat 4 [fd 80 rt 90]
? rt 45 pu fd 10 pd setpc 2
? fillin
```

First   First *object*

Output the first element of an object.

```
? first [book pen ruler]
BOOK

? first "fish
F

? first 1234
1

? first first [red green blue]
R
```

This operation works for both lists and words. If the input is a list, the output of **First** will be the first element of the list. If the input is a word, the first character is output. Keep in mind that numbers are words as well.

It is an error for the input object to be empty:

```
? first []
FIRST does not like [] as input
```

FirstPut

Output the first object combined to the front of the second.

? firstput "book [pen ruler]
[BOOK PEN RULER]

? firstput "f "ish
FISH

? firstput 1 234
1234

FirstPut outputs a new object containing the second object appended to the first. This operation works for both lists and words. If the second input is a list, the first input will be added to the front of the list. If the second input is a word, the letters of the first word will be added to the word. Keep in mind that numbers are words as well.

Forward

Move the turtle forward by a distance.

? forward 40

If the pen is down, draw a line on the graphics screen.

Forward will move the pen in the direction of the current turtle heading. It will not affect the turtle heading.

? clearscreen pendown
? forward 40
? pos
[0 40.0]

```
? right 90
? forward 20
? pos
[20.0 40.0]
```

Back is the reverse operation of **Forward**:

```
? home
? forward 40 back 40
? pos
[0 0]
```

FullScreen

Display the graphics window only.

```
? fullscreen
```

The text window will disappear and the full display area will become available for graphics.

To bring back the text window use the **TextScreen** or **SplitScreen** commands or type CTRL-T.

GetProp

GetProp *name property*
GProp *name property*

Output a specified property of a name.

```
? putprop "william "age 12
? getprop "william "age
12
```

GetProp outputs a property previously input with **PutProp**. The first input is the variable name to which the property is related. The second input is the name of the property being accessed. This name must be the identical to that specified with **PutProp**.

```
? putprop "william "height [60 inches]
? getprop "william "height
[60 inches]
```

Go to the specified label.

Go transfers control back to the line following a matching **Label** command. The input to **Go** is a word which matches that of the **Label** command.

Go can only be used in a procedure and both the **Go** and the **Label** commands must be in the same procedure.

```
to yes.no
print ''y\/n?
label ''again
make ''c readchar
if :c = ''Y [output ''true]
if :c = ''N [output ''false]
print [please answer y or n]
go ''again
end
```

There are often better ways to obtain the same results without using **Go**.

GraphicsType   GraphicsType *object*
                                               GrType *object*

Print text to the graphics screen.

```
? clearscreen
? graphicstype [This is home.]
```

GraphicsType is similar to **Type**, but prints its input to the graphics screen rather than to the text window. Like other graphics commands, the text is drawn with the current pen color at the current pen position. The pen's state (up, down, erase, and reverse) also affects the text printed.

The pen position is not affected by this command. Performing the same command again will strike over the previous text.

This example will print the the word "HOME" in two colors and then erase it:

```
? cs pendown
? setpc 1   graphicstype [home]
? setpc 2   graphicstype [home]
? penerase graphicstype [home] pendown
```

This example prints the position of the pen in several places:

```
? home pendown
? repeat 5 [fd 20 grtype pos]
```

Heading

Output the heading angle of the turtle.

```
? clearscreen
? heading
0.
```

Heading outputs the turtle's direction as an angle measured clockwise from straight up. This angle will range from zero up to (but not including) 360 degrees. Zero is straight up.

It is the turtle heading that determines the direction of lines drawn with the graphics line drawing commands **Forward** and **Back**.

The turtle heading can be modified with **Right, Left, SetHeading,** and **Towards**. The heading is returned to zero with **Home** and **ClearScreen**.

```
? right 90
? heading
90.0

? left 40
? heading
50.0
```

```
? setheading 192
? heading
192.0
```

Remove the turtle pointer from the screen.

```
? hideturtle
? showturtle
```

HideTurtle makes the turtle invisible, but does not affect its drawing abilities. This operation is useful for programs that do not need nor desire the turtle pointer.

The **ShowTurtle** commands can be used to make the turtle visible again. **Shown?** will indicate whether the turtle is visible or not.

```
? hideturtle
? shown?
FALSE
```

```
? showturtle
? shown?
TRUE
```

Center the turtle on the screen and zero its heading.

```
? home
? heading
0.
? pos
[0 0]
```

This command moves the turtle to the center of the graphics screen [0 0] and sets its heading to zero (straight up). This action does not change the state of the pen (up, down, erase, reverse) or its color.

If pred is TRUE, execute the first list, else execute the second.

```
? if "true [print "hi]
HI

? if "false [print "hi] [print "bye]
BYE
```

If conditionally selects and executes lists. The first input must be a predicate that returns either TRUE or FALSE. If it is TRUE, the first list is executed. If it is FALSE, then the second list will be executed if it is present. If the second list is not present, nothing is done and control goes to the next expression.

```
? if 3 > 1 [print "greater] [print [not greater]]
GREATER

? showturtle
? if shown? [print [turtle is visible]]
TURTLE IS VISIBLE
```

Notice that the second and third inputs must be a list. If you forget this, Logo will evaluate these inputs and expect their results to be a list. This may cause a rather confusing error message to occur. For example:

```
? if 2 > 1 print "greater
GREATER
PRINT did not output to IF
```

The **Print** command was executed, and its result was input to the **If** (which was expecting a list to execute). Because **Print** does not supply an output to **If**, the error occurs.

If a **Test** was FALSE, execute the list.

```
? test 1 > 2
? iffalse [print [not greater]]
NOT GREATER
```

IfFalse conditionally executes a list if the result of the most recent **Test** command was FALSE. If the test was TRUE, control is passed to the next expression.

IfTrue

IfTrue *run-list*
IfT *run-list*

If a **Test** was TRUE, execute the list.

```
? test 0 < 10
? iftrue [print [less than]]
LESS THAN
```

IfTrue conditionally executes a list if the result of the most recent **Test** command was TRUE. If the test was FALSE, control is passed to the next expression.

Integer

Integer *number*
Int *number*

Output the integer part of a number.

```
? integer 3.7
3
```

```
? integer -8.5
-8
```

This operation removes the decimal portion of a number and returns just its integer value. The number is not rounded to the nearest integer (see **Round**).

Item

Item *number object*

Output an element of an object.

```
? item 2 [book pen ruler]
pen
```

```
? item 3 "fish
S

? item 1 4321
4

? item 2 item 1 [red green blue]
E
```

This operation works for both lists and words. The first input indicates the position of the element within the second input.

There must be enough elements in the object or an error will occur:

```
? item 5 "fish
Too few items in FISH
```

Key?   Key?
                                                                     KeyP

Output TRUE if there is a character ready to be read.

```
? key?
FALSE
```

Key? will output TRUE whenever a character is waiting to be read from the keyboard. The output will remain TRUE until all waiting characters have been read. Characters are read from the keyboard with **ReadChar** and **ReadList**.

```
? repeat 300 [pr key?] if key? [pr readchar] [pr [no char]]
FALSE
FALSE
FALSE
TRUE('X' key pressed)
TRUE
. . .

X
```

KeyP is the old way of spelling **Key?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Label   Label *name*

Associates a name with a line in a procedure for use with **Go**.

Label marks a point to which control can be transferred with a **Go** command. The input is a word used to identify this point in your program. **Go** must use this same word to transfer control to this point.

Label can only be used in a procedure and both the **Go** and the **Label** commands must be in the same procedure.

```
to yes.no
print "y\ /n?
label "again
make "c readchar
if :c = "Y [output "true]
if :c = "N [output "false]
print [please answer y or n]
go "again
end
```

There are often better ways to perform the same results without using **Go** and **Label**.

Last   Last *object*

Output the last element of an object.

```
? last [book pen ruler]
RULER

? last "fish
H
```

```
? last 1234
4

? last last [red green blue]
E
```

This operation works for both lists and words. If the input is a list, the output of **Last** will be the last element of the list. If the input is a word, the last character is output. Keep in mind that numbers are words as well.

It is an error for the input object to be empty:

```
? last []
LAST does not like [] as input
```

LastPut   LastPut *object object*
                                                  LPut *object object*

Output the first object combined to the end of the second.

```
? lastput "ruler [book pen]
[BOOK PEN RULER]

? lastput "h "fis
FISH

? lastput 4 123
1234
```

LastPut outputs a new object containing the first object appended to the second. This operation works for both lists and words. If the second input is a list, the first input will be added to the end of the list. If the second input is a word, the letters of the first word will be added to the end of the second word. Keep in mind that numbers are words as well.

Left

Rotate the turtle counter-clockwise.

? left 45

Left turns the turtle heading to the left (counter-clockwise). The input specifies the number of degrees to turn. The X-Y position of the turtle is not affected.

? home
? left 45
? heading
315.0

? clearscreen
? repeat 6 [forward 60 left 144]

List

Output a list containing the input objects as elements.

? list ''apple ''grape
[APPLE GRAPE]

? list 123 456
[123 456]

? list [sean alex] [cindy caryn]
[[SEAN ALEX] [CINDY CARYN]]

This operation creates a new list with each of the input objects as elements.

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

? (list ''apple ''orange ''banana ''grape)
[APPLE ORANGE BANANA GRAPE]

List?   List? *object*
                                                                    ListP *object*

Output TRUE if an object is a list.

? list? [peach grape]
TRUE

? list? "grape
FALSE

? list? 123
FALSE

List? will output TRUE whenever its input is a list.
Other types of inputs will output a FALSE.

ListP is the old way of spelling **List?**. We include it for
compatibility. The "P" stands for the word
"predicate" which indicates that a TRUE or FALSE
will be output.

Load   Load *file*

Load a file into the workspace or a package.

? load "init

Load requests Logo to read and execute expressions
from a disk file rather than from the keyboard. The
file will be read until either the end is reached or an
error occurs. It can contain procedure definitions,
variable bindings, and direct commands and
expressions.

The file name can also include a device name and
directory path. For example:

? load "df0:logo\/test

If the file being loaded was saved as a package (see
Save), it will be loaded as a package (see **Package**
below).

If the file cannot be found, an error will occur:

```
? load "banana
File BANANA not found
```

Local   Local *name*
                                                   (Local *name name* . . .)

Make a name local to a procedure.

This command makes a variable private to a
procedure. With it new variables can be defined that
only exist during the execution of the procedure in
which they were declared to be local. This is a good
way to use variable names that have the same name
either globally or in another procedure. It also offers a
way to "hide" data within a procedure, and reduce
the overall complexity of a program.

Local can be used only in a procedure and it must be
performed prior to the use of the local variable (for it
to take effect).

```
to "example
local "angle
make "angle 45
. . .
end
```

When a procedure stops or outputs a value, the local
variable values are lost. Any existing variables using
the same name will retain their previous values.
Input names are also local variables.

Normally only one input can be supplied to this
operation. However, when enclosed in parentheses,
the operation will accept any number of inputs.

```
to "example
(local "angle "distance "hue "state)
. . .
end
```

Output the welcome banner for Amiga Logo.

```
? logo

AMIGA LOGO, Version 1.00
Copyright (C) 1989 Commodore Amiga, Inc.
Copyright (C) 1989 Carl Sassenrath
All rights reserved.
```

Make   Make *name object*

Make a variable and give it a value.

```
? make "one 1
```

This command creates a new variable with a specified name and binds a value to it.

Once created a variable can be accessed by placing a colon ':' (called "dots") in front of its name.

```
? print :one
1
```

The effect of a **Make** depends on the environment in which it was executed. If a **Make** specifies the name of a global variable (which has not since been made local), a new variable is created (with the same name), and the old value can no longer be referenced with this name. However, the old value can be retained by using **Local** (see above) prior to the **Make**.

Member?   Member? *object object*
                                                 MemberP *object object*

Output TRUE if the first object is a member of the second.

```
? member? "grape [apple orange grape]
TRUE
```

```
? member? ''g ''grape
TRUE

? member? 3 1234
TRUE
```

Member? detects the presence of a particular element within an object. The first input is the element to find. The second is either a list or a word. For lists a TRUE will be output if the element is present in the list. For words, a TRUE will be output if the character is present in the word.

Note that **Member?** only searches the first level of a list. It does not search individual elements of a list. For example:

```
? member? ''grape [apple orange [grape banana]]
FALSE
```

Grape is not a member of this list even though there is an element that contains the word grape.

MemberP is the old way of spelling **Member?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Mouse   Mouse

Output the X,Y position of the mouse.

```
? mouse
[310 71]   (depends on where the mouse is)
```

Mouse returns the position of the Amiga mouse pointer within the screen. The mouse button does not need to be held down for this to work.

Name   Name *object name*

Attach a value to a variable name.

```
? name 5 ''blue
```

This command creates a new variable with a specified name and binds a value to it. **Name** is identical to **Make**, except the order of its inputs is reversed.

Once created a variable can be accessed by placing a colon ':' (called "dots") in front of its name.

? setpencolor :blue

? pencolor
5

The effect of a **Name** depends on the environment in which it was executed. If a **Name** specifies the name of a global variable (which has not since been made local), a new variable is created (with the same name), and the old value can no longer be referenced with this name. However, the old value can be retained by using **Local** (see above) prior to the **Name**.

Name?   Name? *object*
                                                   NameP *object*

Output TRUE if an object has a value.

? make "blue 5
? name? "blue
TRUE

? name? 120
FALSE

Name? will output TRUE whenever its input is a name representing a value (a variable).

NameP is the old way of spelling **Name?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Output the number of free memory nodes.

```
? nodes
660
```

The value returned indicates how much free memory is available for use by Logo. This value will fluctuate depending on many factors. Occasionally Amiga Logo will collect garbage memory and reuse it (see **Recycle**). Whenever this happens the number of free nodes will increase.

Some of the memory nodes included in the output from this operation are also available to other programs on the Amiga. If you are running other programs at the same time, they may greatly reduce the number of free nodes available to Logo.

Not   Not *pred*

Output TRUE when FALSE, and FALSE when TRUE.

```
? not "true
FALSE

? not "false
TRUE
```

The input to **Not** is called a *predicate*. It must be either TRUE or FALSE. **Not** will output the opposite value.

Not is quite useful for reversing the logic in comparisons:

```
? not (1 > 10)
TRUE
```

would be equivalent to "is 1 less than or equal to 10?"

If you attempt to input something other than TRUE or FALSE, Logo will inform you that you made a mistake:

```
? not 3
3 is not true or false
```

Number?   Number? *object*
                                                NumberP *object*

Output TRUE if the input is a number.

```
? number? [peach grape]
FALSE

? number? "grape
FALSE

? number? 123
TRUE
```

Number? will output TRUE whenever its input is a number. Other types of inputs will output a FALSE.

NumberP is the old way of spelling **Number?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Or   Or *pred pred*
                                                (Or *pred pred* . . .)

Output TRUE if any inputs are TRUE.

```
? or "true "true
TRUE

? or "true "false
TRUE

? or "false "false
FALSE
```

The inputs to **Or** are called *predicates*. They must be either TRUE or FALSE.

For example, the comparison of two numbers outputs either TRUE or FALSE, so this could be used as input to **Or**:

```
? or (1 > 10) (40 > 20)
TRUE
```

If you attempt to input something other than TRUE or FALSE, Logo will inform you that you made a mistake:

```
? or 3 4
3 is not true or false
```

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

```
? ( or "false "true "false "true )
TRUE
```

Here is an example of how **Or** might be used in graphics:

```
? setpos [45 72]
? make "x first pos
? make "y last pos
? (or (:x < 0) (:x > 160) (:y < 0) (:y > 100))
FALSE
```

Output   Output *object*
                                                                OP *object*

Return an object as the output from a procedure.

This operation can only be used in a procedure. When it is executed, the current procedure will be terminated, and the input to **Output** will become the procedure's output. Any type of object can be output: numbers, words, lists, predicates, etc.

For example, the procedure:

```
TO FOUR
OUTPUT 4
END
```

will always output a four:

```
? four
4
```

Package   Package *package name*
                                                               Package *package name-list*

Put the named procedures or variables into a
package.

```
? package ''work ''my.proc
```

```
? package ''work [my.proc my.var1 my.var2]
```

Package lets you bundle together related procedures
and variables to be saved, loaded, printed, edited,
buried (hidden), and erased as a single unit. The first
input is the name of the package. The second is a
procedure or variable name (or a list of them).

If a name being packaged already belongs to another
package, it will be transferred to this package.

Packages are helpful for writing utilities that you use
in a number of other programs.

PackageAll   PackageAll *package*
                                                               PkgAll *package*

Put all unpackaged procedures and variables into a
package.

```
? packageall ''work
```

PackageAll performs a **Package** command over all
procedures and variables in the workspace. See
Package for more detail.

Suspend execution of the current procedure.

This command can only be used in a procedure. When it is executed, the current procedure will be saved, and control will return to the Logo command level. Pressing CTRL-Z during the execution of a procedure has the same effect.

To continue execution of the paused procedure, type **Continue**.

Output a list of the pen state and the pen color.

```
? setpencolor 1
? penerase
? pen
[PENERASE 1]
```

The output from **Pen** is a two element list. The first element describes the state of the pen (up, down, erase, reverse), and the second indicates its color.

Output the color number of the pen.

```
? pencolor
1
```

PenColor outputs a color number that indicates the current color of the pen. The actual pen color depends on the color setting for this number (see **RGB**). The number may range in value from zero to 30.

The color of the pen can be set with the **SetPenColor** command.

```
? setpencolor 2
? pencolor
2
```

When Logo is started, it establishes certain colors.
The first 4 colors are:

0   *(background)*
1   *white*
2   *green*
3   *violet*

These colors were picked to be compatible with most
Logo textbooks. They can be modified with the
SetRGB command (below). See the **RGB** entry for a
complete list of colors.

PenDown

<div align="right">

PenDown
PD

</div>

Put the pen into its draw state.

```
? pendown
```

PenDown starts the turtle drawing. The next time the
turtle is moved, a line will be drawn in the current
pen color.

The **PenDown** drawing mode can be stopped with
the **PenUp** command. Other commands **PenErase**
and **PenReverse** also stop **PenDown**.

```
? home penup
? forward 30
? pendown
? forward 30
```

PenErase

<div align="right">

PenErase
PE

</div>

Put the pen into its erasing state.

```
? penerase
```

PenErase puts the pen into an erase mode. The next time the turtle is moved, it will erase whatever it passes over.

The **PenErase** mode can be stopped with **PenUp** , **PenDown**, or **PenReverse**.

```
? home pendown
? forward 30
? penerase
? back 20
```

PenReverse
<div align="right">PenReverse
PX</div>

Put the pen into its reverse state.

```
? penreverse
```

PenReverse puts the pen into its reversing mode. When the pen moves it will complement the color it passes over: that is, it will draw where there are no lines, and erase where there are lines (of the same color).

The **PenReverse** mode can be stopped with **PenUp** , **PenDown**, or **PenErase**.

```
? home
? repeat 4 [pendown fd 20 penup fd 10]
? penreverse
? back 80
```

PenUp
<div align="right">PenUp
PU</div>

Put the pen into its no-draw state.

```
? penup
```

PenUp stops the turtle from drawing, erasing, or reversing. When the turtle moves, the pen does nothing.

PenUp turns off **PenDown**, **PenErase**, and **PenReverse**.

? repeat 12 [penup fd 10 rt 30 pendown fd 10]

Position

Output the position of the turtle as an X,Y list.

? home
? position
[0 0]

Position outputs the current position of the turtle as a list of two numbers. The first number is the x position; the second is the y position.

? home
? forward 20
? position
[0 20.0]

? right 90 fd 30
? position
[30.0 20.0]

Primitive?

Primitive? *object*
PrimitiveP *object*

Output TRUE if the input is the name of a Logo primitive.

? primitive? "pen
TRUE

? primitive? "one
FALSE

Primitive? will output TRUE whenever its input is a Logo primitive. Other types of inputs will output a FALSE.

PrimitiveP is the old way of spelling **Primitive?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Print

Print *object*
Pr *object*
(Print *object object*...)

Print an object, then start a new line.

```
? print "hello
HELLO

? print [this is logo]
THIS IS LOGO
```

Print prints an object to the text window then starts a new line. When lists are printed, their outermost brackets will be removed.

The **Type** command is similar to **Print**, but does not start a new line. The **Show** command is also similar to **Print**, but will include the outermost brackets when printing.

Normally only one input is supplied to **Print**. However, when enclosed in parentheses, it will be applied to any number of inputs.

```
? (print "apple 123 [orange peach])
APPLE 123 ORANGE PEACH
```

PrintOut

PrintOut *name*
PrintOut *name-list*

PO *name*
PO *name-list*

Print the named procedures and variables.

```
? define "south [ [] [setheading 180]]
? printout "south
TO SOUTH
SETHEADING 180
END
```

```
? make "red 7
? printout "red
MAKE "RED 7
```

PrintOut prints the name and definition for a procedure, and the name and value for a variable.

PrintOut accepts the name of a procedure or variable as input. It will also accept a list of procedures and variable names. Supplying no input will print an error message.

PrintOutAll

PrintOutAll
PrintOutAll *package*
PrintOutAll *package-list*

POAll
POAll *package*
POAll *package-list*

Print everything in the workspace, a package, or packages.

PrintOutAll is similar to **PrintOut** but handles an entire package containing many definitions and variables.

PrintOutAll accepts the name of a package or a list of packages as input. Supplying no input will default to printing all unburied procedures and variables in the workspace.

PrintOutNames

PrintOutNames
PrintOutNames *package*
PrintOutNames *package-list*

PONs
PONs *package*
PONs *package-list*

Print all variables in the workspace, a package, or packages.

PrintOutNames is similar to **PrintOutAll** but handles just the variables within a package. **PrintOutNames**

accepts the name of a package or a list of packages as input. Supplying no input will default to printing all unburied variables in the workspace.

PrintOutProcs

PrintOutProcs
PrintOutProcs *package*
PrintOutProcs *package-list*

POPs
POPs *package*
POPs *package-list*

Print all procedures in the workspace, a package, or packages.

PrintOutProcs is similar to **PrintOutAll** but handles just the procedures within a package. **PrintOutProcs** accepts the name of a package or a list of packages as input. Supplying no input will default to printing all unburied procedures in the workspace.

PrintOutTitles

PrintOutTitles
PrintOutTitles *package*
PrintOutTitles *package-list*

POTS
POTs *package*
POTs *package-list*

Print procedure titles in the workspace, package, or packages.

PrintOutTitles is similar to **PrintOutProcs** but prints just the title line (the first line of a **To** definition) of procedures within a package. **PrintOutTitles** accepts the name of a package or a list of packages as input. Supplying no input will default to printing all procedure titles in the workspace.

Product   product *number number*
                                          (product *number number...*)

                                                   number * *number*
                                                    * *number number*
                                          (* *number number...*)

Output the product of the input numbers.

See the description for ''*''.

PropList   PropList *name*
                                                          PList *name*

Output a list of all properties associated with a name.

    ? putprop ''william ''age 12
    ? putprop ''william ''height [60 inches]
    ? proplist ''william
    [HEIGHT [60 INCHES] AGE 12]

PropList outputs a list of properties previously input
with **PutProp**. The input is the variable name to
which the properties are related.

Note that the order of items output in the property
list is not the same as the order in which they were
added.

PutProp   PutProp *name property object*
                                               PProp *name property object*

For a name, create a property with a given value.

    ? putprop ''bill ''address [68000 guru way]
    ? putprop ''bill ''city [amigaville]
    ? putprop ''bill ''state [ca]
    ? putprop ''bill ''country [usa]
    ? proplist ''bill
    [COUNTRY [USA] STATE [CA] CITY [AMIGAVILLE]
    ADDRESS[68000 GURU WAY]]

PutProp creates a property for a variable name and
binds a value to it. If the variable name already has
this property, the old value will be replaced with the
new one.

The property can be accessed with **GetProp** and erased with **RemProp**.

Quit

Quit
Exit

Exit from Logo and return to Workbench or CLI.

? quit

Quit will terminate this session of Logo and return you to the Workbench, the CLI, or whatever the program was that started Logo.

Prior to exiting Logo you may want to save your changes. See the **Save** and **SaveFile** commands below.

Quotient

quotient *number number*
(quotient *number number . . .*)

number / *number*
/ *number number*
(/ *number number . . .*)

Output the quotient of the input numbers.

See the description for "/".

Random

Random *number*

Output a random number.

? random 10
4

?random 100
79

This operation outputs a random integer number. The input specifies the upper limit of the number. The output will be greater than or equal to zero but less than the input number.

? repeat 100 [home rt random 360 fd random 80]

Reference   167

Random will always generate the same sequence of random numbers each time Logo is started. This sequence can be started again with the **Rerandom** command.

ReadChar

<div align="right">ReadChar
RC</div>

Output a character typed on the keyboard.

```
? readchar
A ("a" was typed)
```

ReadChar outputs the next character typed on the keyboard. If there are no characters ready to be read, **ReadChar** waits until one is typed.

ReadChar does not require that the user type a RETURN after each character.

Characters being read with **ReadChar** are not printed to the screen when they are typed. You must use a **Print** to echo them back to the user.

```
? repeat 4 [print readchar]
P          ("p" was typed)
E          ("e" was typed)
A          ("a" was typed)
R          ("r" was typed)

? if readchar = "Y [print "yes] [print "no]
yes ("y" was typed)
```

ReadList

<div align="right">ReadList
RL</div>

Output a list from a line typed on the keyboard.

```
? readlist
rj caryn [alex robyn]      (typed on keyboard)
[RJ CARYN [ALEX ROBYN]]
```

ReadList accepts a line of characters from the keyboard, converts them to a list, and outputs the list.

The keyboard characters are converted to a list in the same fashion as all Logo text entry (see tutorial section).

Characters will be printed as they are typed and the line can be edited just like any other Logo line. A RETURN is required to enter the line.

```
? if number? first readlist [print "number]
1234 (typed on keyboard)
NUMBER
```

Recycle   Recycle

Free garbage memory nodes for reuse.

```
? recycle
```

Recycle forces Logo to reuse old garbage memory nodes. This normally happens automatically when Logo exhausts its supply of nodes, but **Recycle** forces it to happen immediately.

Depending on how much memory is involved, a recycle may take some time. It is usually seen as a pause during the execution of a procedure. It can be very noticeable during graphics drawing sequences. Using **Recycle** you can force a garbage collection at a more convenient time and often make the pauses less visible.

Remainder   Remainder *number number*

Output the integer remainder of a divide.

```
? remainder 10 2
0

? remainder 5 2
1
```

Remainder performs an integer division and outputs the remainder of the division. The first input is divided by the second.

This function is often called the **Mod** function in other languages.

RemProp   RemProp *name property*

Remove a property and its value from a name.

```
? putprop "william "age 12
? getprop "william "age
12
? remprop "william "age
```

RemProp removes a property previously input with **PutProp**. The first input is the variable name to which the property is related. The second input is the name of the property being removed. This name must be identical to that specified with **PutProp**.

Repeat   Repeat *number run-list*

Execute a list a number of times.

```
? repeat 3 [print "hello]
HELLO
HELLO
HELLO
```

Repeatedly execute the same list a number of times. The first input is an integer which specifies the number of time to repeat. The second input is the list to execute.

Rerandom   Rerandom

Restart the sequence of random numbers.

```
? rerandom
? repeat 5 [(type random 100 " )]
47 79 88 40 95
```

```
? rerandom
? repeat 5 [(type random 100 " )]
47 79 88 40 95
```

Rerandom is used to reset the random number
sequence to that found when Logo is first started.
After executing **Rerandom** the same inputs to
Random will produce the same outputs as before.

RGB   RGB *color-number*

Output a list of the RGB components of a color
number.

```
? rgb 1
[14 14 14]
```

Given a valid color number as input, this operation
will output a list of three numbers which are the red,
green, and blue (RGB) components of the color.

Color numbers are used to set the screen colors for
the pen and background. They may range in value
from one to 30. Color zero cannot be modified
directly, you must assign it to a color number, then
modify the color number.

The RGB components indicate how much of red,
green, and blue are mixed to make a given color. Each
of the RGB components should be a number between
0 and 15. So:

[12 7 3]

would be 12 units of red, 7 of green, and 3 of blue.
The resulting color would be orange-gold.

When Logo is started, it establishes RGB color values for all color numbers. The colors are:

0 (background)
1 white
2 green
3 violet
4 orange
5 blue
6 black
7 red
8 yellow
9 grey
10 pink
11 navy blue
12 peach
13 brown
14 forest green
15 cyan
16 dark brown
17 olive green
18 mustard
19 bright red
20 bright green
21 bright blue
22 dark purple
23 dark grey
24 dark blue
25 hot pink
26 sky blue
27 royal blue
28 aqua green
29 silver grey
30 lawn green
31 soft pink

These colors were picked to be compatible with most Logo textbooks. They can be modified with the **SetRGB** command.

Rotate the turtle clockwise.

    ? right 45

Right turns the turtle heading to the right (clockwise). The input specifies the number of degrees to turn. The X-Y position of the turtle is not affected.

    ? home
    ? right 45
    ? heading
    45.0

    ? clearscreen
    ? repeat 6 [forward 60 right 144]

Round   Round *number*

Output a number rounded to the nearest integer.

    ? round 10.9
    11

    ? round 10.35
    10

    ? round -3.45
    -3

Round outputs the input number rounded to the nearest integer.

Run   Run *run-list*

Execute a list.

    ? run [print "hello]
    HELLO

Execute the list given as input. Output a value if executing the list outputs a value.

This operation is useful for running Logo code that was itself created by a Logo program. For example:

```
? make "box.side [fd 40 rt 90]
? run (list "repeat 4 :box.side)
```

Save

Save the workspace or a package to a file.

```
? save "work
```

Save requests Logo to save the workspace or a package to a disk file. All procedure definitions and variable bindings will be saved.

The file name can also include a device name and directory path. For example:

```
? save "df0:logo\/test
```

Save also accepts an optional package as input. If specified, all procedures and variables defined in the package (instead of the workspace) will be saved. The bury state of the package will also be saved.

If the file already exists, Amiga Logo will remove the old file first, then save the new one. No error message will occur.

SaveEdit

Save the editor buffer to a file.

```
? saveedit "work
```

SaveEdit requests Logo to save the entire editor buffer to a disk file. Everything within the editor buffer will be saved, including commands, comments, and incomplete lines.

SaveEdit is a useful command for saving text edited with the EditFile command. For example:

```
? editfile "s:startup_-sequence
(make changes, then exit editor with CTRL-G)
? saveedit "s:startup_-sequence
```

It is also useful for adding direct commands to a file that has already been saved:

```
? save "work
? editfile "work
(add some commands to the file, exit with CTRL-G)
? saveedit "work
```

Say   Say *word*
                                                           Say *list*

Make the Amiga talk.

```
? say "hello
```

Say will generate Amiga speech. You can give it either a word or list for input.

```
? say [there is no place like home]
```

To work correctly, the **Say** command requires Workbench 1.3 and the Speak device must be mounted (see DOS manual).

To hear speech, you must have an amplifier or monitor (with built-in amplifier) connected to the Amiga audio outputs. The Amiga itself does not contain a speaker.

Say will output garbled speech when in 16 color 640 mode.

Scrunch   Scrunch
                                                           Aspect

Output the aspect ratio of the screen.

See the description for **Aspect**.

Sentence

Output a list containing the words of the input objects.

```
? sentence "apple "grape
[APPLE GRAPE]

? sentence 123 456
[123 456]

? sentence [cindy caryn] [sean alex]
[CINDY CARYN SEAN ALEX]
```

This operation creates a new list with each of the input objects as elements. It is similar to the **List** operation, but strips the outer brackets off any inputs that are lists.

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

```
? (sentence "apple "orange [banana grape])
[APPLE ORANGE BANANA GRAPE]
```

SetAspect

Set the aspect ratio of the screen.

```
? setaspect 1
```

SetAspect accepts as input the desired aspect ratio of the graphics screen. This ratio controls the scale of the Y axis compared with the X axis. It is the number of units in the Y direction for each unit in the X direction.

SetAspect is provided to compensate for different screen resolutions and monitor brands. If your circles look like ellipses and squares look like rectangles, the aspect ratio can correct these.

For example, to make each vertical unit half the size of a horizontal unit, an aspect ratio of 0.5 would be used:

? setaspect .5

The current screen aspect ratio can be found with **Aspect**.

SetScrunch is the old word for **SetAspect**. It is included for compatibility with other Logos.

SetBackground

SetBackground *color-number*
SetBg *color-number*

Set the screen to the color of a color number.

? setbackground 4

The input is a color number. This number is used as an index into the screen color table. The actual background color displayed depends on the color setting for this color number. For more information see the explanation for **RGB**.

Note that color number zero is the same as the background. Setting the background to zero does nothing. Amiga Logo restricts RGB operations that are performed on color zero directly.

SetCursor

SetCursor *[column row]*

Set the text cursor position.

? setcursor [0 10]
? cursor
[0 12]

SetCursor accepts as input a list containing the column and row positions of the cursor within the text window, and moves the cursor to that position.

The upper left cursor position of the text window is [0 0]. The maximum values for the column and row of the text cursor will depend on the size of the text window.

The position of the cursor can be found with **Cursor**.

SetHeading   SetHeading *angle*
                                                       SetH *angle*

Set the heading angle of the turtle.

```
? home
? setheading 45
? heading
45.0
```

SetHeading sets the turtle's direction as an angle measured clockwise from straight up. The input angle can range from zero (straight-up) to less than 360 degrees. Angles greater than 360 wrap back to zero at 360.

```
? setheading 400
? heading
40.0
```

It is the turtle's heading that determines the direction of lines drawn with the graphics line drawing commands **Forward** and **Back**.

The turtle heading can also be modified with **Right**, **Left**, and **Towards**. The heading is returned to zero with **Home** and **ClearScreen**.

```
? home
? right 90
? heading
90.0
```

```
? left 45
? heading
45.0

? setheading 192
? heading
192.0
```

SetPen   SetPen *[pen-state color-number]*

Set the pen to a new state and color.

```
? setpen [penup 5]
? pen
[PENUP 5]
```

SetPen accepts a two element list as input. The first
element is the state of the pen and the second
indicates its color. The state of the pen must be one of
the four words: **PenUp**, **PenDown**, **PenErase**, or
PenReverse, and should not be abbreviated.

The list input to **SetPen** is the same as the list output
from **Pen.**

SetPenColor   SetPenColor *color-number*
                                                          SetPC *color-number*

Set the pen to the color of a color number.

```
? setpencolor 3
```

SetPenColor accepts as input a color number that
indicates the desired color of the pen. The actual pen
color depends on the color setting for this number
(see **RGB**). The number may range in value from zero
to 30.

The current color of the pen can be found with the
PenColor command.

```
? pencolor
3
```

When Logo is started, it establishes certain colors. The first 4 colors are:

0   *(background)*
1   *white*
2   *green*
3   *violet*

If you input a number greater than the number of colors selected, Amiga Logo does a modulo to determine a selectable option. This means that the number is divided by the number of colors, with the remainder determining the number of a selectable color.

These colors were picked to be compatible with most Logo textbooks. They can be modified with the **SetRGB** command.

SetPosition   SetPosition *[x y]*
                                                     SetPos *[x y]*

Move the turtle to an X,Y position.

    ? setposition [30 60]
    ? position
    [30.0 60.0]

SetPosition accepts as input a new turtle position as a list of two numbers. The first number is the x position; the second is the y position.

The current position of the turtle can be found with **Position**.

SetRGB   SetRGB *color-number [red green blue]*

Assign a color number to an RGB color.

    ? setrgb 5 [12 7 3]
    ? rgb 5
    [12 7 3]

```
? clearscreen
? setpencolor 5
? repeat 6 [forward 60 right 144]
```

SetRGB sets the red, green, and blue (RGB) components of a screen color. The first input is the color number to set. The second input is a list of three numbers which are the red, green, and blue (RGB) components of the color.

Color numbers are used to set the screen colors for the pen and background. They may range in value from one to 30.

The **RGB** operation can be used to get the current RGB settings for a color number.

The RGB components indicate how much of red, green, and blue are mixed to make a given color. Each of the RGB components should be a number between 0 and 15. So:

[12 7 3]

would be 12 units of red, 7 of green, and 3 of blue. The resulting color would be orange-gold.

When Logo is started, it establishes RGB color values for all color numbers. The first 4 colors are:

0   *(background)*
1   *white*
2   *green*
3   *violet*

These colors were picked to be compatible with most Logo textbooks. See the **RGB** entry for a complete list of Logo colors.

SetScrunch   SetScrunch *number*
                                                       SetAspect *number*

Set the aspect ratio of the screen.

See the description for **SetAspect**.

SetX   SetX *x*

Set the X position of the turtle.

> ? setx 40

The input is a number indicating the X coordinate position. This is the position of the turtle horizontally from the home position. The positive direction is to the right; negative is to the left. The turtle's Y position does not change.

If the screen is in the **Window** mode, the position returned might be greater than can be displayed on the screen.

SetY   SetY *y*

Set the Y position of the turtle.

> ? sety 40

The input is a number indicating the Y coordinate position. This is the position of the turtle vertically from the home position. The positive direction is to the right; negative is to the left. The turtle's X position does not change.

If the screen is in the **Window** mode, the position returned might be greater than can be displayed on the screen.

Show   Show *object*
                                                   (Show *object object* . . .)

Print an object, then start a new line.

> ? show ''hello
> HELLO
>
> ? show [this is logo]
> [THIS IS LOGO]

Show prints an object to the text window then starts a new line. It is similar to **Print**, but it includes the outermost brackets when printing a list.

Normally only one input is supplied to **Show**; however, when enclosed in parentheses, it will be applied to any number of inputs.

```
? (show "apple 123 [orange peach])
APPLE 123 [ORANGE PEACH]
```

Shown?

Output **True** if the turtle pointer is showing.

```
? showturtle
? shown?
TRUE
```

Shown? will output TRUE as long as the turtle is visible on the graphics screen. It will output a FALSE if the turtle is hidden (see **HideTurtle**).

```
? hideturtle
? if not shown? [print [turtle is hiding!]
TURTLE IS HIDING!
```

ShownP is the old way of spelling **Shown?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

ShowTurtle

Display the turtle pointer.

```
? showturtle
```

ShowTurtle makes the turtle visible (if it's on the screen). The turtle is made invisible with the **HideTurtle** command.

Shown? will indicate whether the turtle is visible or not.

```
? showturtle
? shown?
TRUE
```

If the **Window** mode is set, the turtle may be off the edge of the screen, and it will not be displayed, but **Shown?** will still return TRUE.

Sine

Sine *angle*
Sin *angle*

Output the sine of an angle.

```
? sine 30
0.5
```

Given an input number representing an angle in degrees, **Sine** will output its sine value. The sine of an angle will always be in the range of 1 to 0 to -1.

```
? sine 0
0

? sine 90
1.0

? sine 270
-1.0
```

The sine of an angle is very useful for many types of graphics operations. For example:

```
? clearscreen
? right 75 forward 80 print xpos
? home right 90
? forward (80 * sine 75) print xpos
```

Both lines will be drawn with the same distance in the X direction.

Display both the graphics and text windows.

```
? splitscreen
```

Split screen allows you to see both text and graphics at the same time. The text screen is made smaller, and the graphics can be seen behind it.

When the screen is split, the text window can be moved by dragging its title bar with the mouse. It can also be resized with the sizing gadget in the lower right of the window. (See your Amiga Workbench manual for more information on using windows.)

The **TextScreen** command will return you to a full text screen or the **FullScreen** command will give you the full graphics display.

SqRt   SqRt *number*

Output the square root of a number.

```
? sqrt 25
5.0

?sqrt 2
1.414214
```

SqRt outputs the square root of its input. The square root is that number which when multiplied by itself equals the original input.

A negative input to **SqRt** causes an error:

```
? sqrt -36
Number is out of range
```

The square root operation is useful for calculating distances in graphics procedures.

Return from a procedure.

This operation can only be used in a procedure. When it is executed, the current procedure will be terminated without producing an output. (The **Output** operation lets you perform a **Stop** and produces an output.)

For example, the procedure:

```
TO CHECK.IT :A
IF :A < 0 [STOP]
PRINT :A
END
```

will not print :A if it is less than zero.

Sum   sum *number number*
                                          (sum *number number* . . .)

                                                  number + number
                                                  + number number
                                          (*+ number number* . . .)

Output the sum of the input numbers.

See description for " + ".

Tangent   Tangent *angle*
                                                          Tan *angle*

Output the tangent of an angle.

```
? tangent 45
1.0
```

Given an input number representing an angle in degrees, **Tangent** will output its tangent value.

```
? tangent 0
0
```

```
? tangent 89
57.28996

? tangent 135
-1.0
```

The tangent of an angle is useful for some types of graphics operations.

Test   Test *pred*

Save a condition for later use in **IfFalse** and **IfTrue**.

```
? showturtle
? test shown?
? iftrue [print [turtle is showing]
TURTLE IS SHOWING
```

Test works in conjunction with the **IfTrue** and **IfFalse** commands. The first input must be a predicate that returns either TRUE or FALSE. It is saved by Logo until another **Test** command is executed.

Text   Text *name*

Output a procedure definition as a list of lists.

Text will output a procedure definition as a list. The format of this list is identical to that input to the **Define** command.

The first element is a list of input names to the procedure. These names do not have a colon ':' before them. If there are no inputs, an empty list [ ] is printed.

The rest of the list is the executable part of the procedure definition.

```
? define "segs [[a size] [repeat 8 [rt :a fd :size]]]
? text "segs
[[A SIZE] [REPEAT 8 [RT :A FD :SIZE]]]
```

Display the text window only.

    ? textscreen

Display the text window making it the full size of the screen.

The **SplitScreen** command will let you see both text and graphics. The **FullScreen** command will give you all graphics.

Thing   Thing *name*

Output the value or definition of a name.

    ? make "one 1
    ? thing "one
    1

Thing is used to access the value of a variable. The input is a word which is the name of the variable to access. This input word can be constructed from other words, or can be the value of a variable itself.

For example, a simple variable:

    ? print :one

is equivalent to:

    ? print thing "one

Here is an example of a variable which holds the name of another variable:

    ? make "unity "one
    ? print :unity
    ONE
    ? print thing :unity
    1

Pass control back to the matching **Catch**.

Throw causes control to be passed back to the matching **Catch** statement.

The input is a name used to identify the **Catch** that will receive this throw.

```
to do.this
print "hello
if key? [throw "key.pressed]
do.this
end

? catch "key.pressed [do.this]
HELLO
HELLO
. . .        until a key is pressed.
```

The target **Catch** command should have been executed already, and it must specify the same name. If a catch with the same name cannot be found, an error will occur:

```
? throw "bad.apples
Cannot find catch for BAD.APPLES
```

With **Throw**, control can be passed from your procedure all the way back to the top level (command level) of Logo:

```
throw "toplevel
```

This is provided for compatibility with other Logo implementations. In most cases it is easier to just invoke the **TopLevel** primitive:

```
toplevel
```

to return to the top level of Logo.

To *name*
To *name input*
To *name input input . . .*

Define a procedure; invoke the editor if necessary.

```
? to "proc

TO PROC
END

(press CTRL-C to exit the editor)
```

To defines a new procedure. If invoked interactively
(from the keyboard) Logo will transfer you to the
editor, and allow you to define the procedure. If
invoked from a file, the lines of text following **To** will
be entered as the procedure definition.

The first input is the name of the procedure to define.
It should begin with a quote ''. Any remaining inputs
will become the inputs to the new procedure. These
words should begin with a colon '':''.

For more about using the editor and defining
procedures, see the tutorial section of this manual.

Procedures can also be defined with the **Define**
command.

TopLevel   TopLevel

Pass control back to the top level of Logo.

```
toplevel
```

This command will abort the current procedure and
all other procedures that called it, and then return to
the command level of Logo. This command is
sometimes used with **Throw** to accomplish the same
end.

Towards   Towards *[x y]*

Output the heading angle needed to face an X,Y position.

```
? home
? towards [10 10]
45.0
```

Towards outputs the heading angle in degrees that the turtle needs to face a given position from the current position. The input is an X, Y list specifying the position.

By using **Towards** with **SetHeading**, a new heading can be established without an angle being specified.

Type   Type *object*
                                    (Type *object object* . . .)

Print an object (but do not start a new line).

```
? type "hello type "there
HELLOTHERE

? (type "every "one)
EVERY ONE
```

Type is similar to **Print**. It prints an object to the text window, but does not start a new line (**Print** does start a new line). When lists are printed, their outermost brackets will be removed.

Normally only one input is supplied to **Type**. However, when enclosed in parentheses, it will be applied to any number of inputs.

```
? (type "apple 123 [orange peach])
APPLE 123 ORANGE PEACH
```

Unbury   Unbury *package*

Expose all procedures and names hidden in a package.

This command undoes the effect of a **Bury**. All procedures and variables defined within the specified package will be visible to certain workspace commands.

Bury is a good way to hide procedures and variables that you have finished. Often you will want to save utility procedures for reuse in several different programs, and you don't want all these utility procedures to get in the way of the program you are developing.

The following workspace primitives are affected by **Bury**:

Save	PrintOutTitles	
EditAll	EditNames	EditProcs
EraseAll	EraseNames	EraseProcs
PrintOutAll	PrintOutNames	PrintOutProcs

For additional information, see **Package**.

Version

Output the version number of Amiga Logo.

```
? version
1.0
```

Version outputs the version of Amiga Logo as a decimal number. As Logo is updated over time, this version number will change.

A decimal number is output whose integer part is the system version (indicating major releases), and decimal part is the revision (for minor changes).

```
? (print [major version is] integer version)
MAJOR VERSION IS 1
```

Wait   Wait *number*

Delay execution for a number of 60ths of a second intervals.

? wait 60

Wait will stop Logo execution for a given period. The input specifies the duration of this period in 60ths of a second.

? clearscreen penreverse
? repeat 60 [fd 50 bk 50 rt 6 wait 60]

The **Wait** command also accepts the special word "**Frame** as input. When this is done, Logo will wait until the video circuitry has started its retrace (blanking) before continuing. Using **Frame** helps prevent many of the flicker effects that occur when drawing to the screen.

Window   Window

Give the turtle unrestricted movement.

? window

Normally when the turtle hits the edge of the screen, an error is generated. This command puts the turtle into a special mode where the turtle can be moved anywhere within an imaginary space including completely off the screen.

? window
? clearscreen
? left 30
? forward 10000
? position
[−5000.001 8660.253]

Word   Word *word word*
                                                (Word *word word* . . .)

Output a word composed of the input words combined.

Reference   193

```
? word "amiga "logo
AMIGALOGO

? word 12 34
1234
```

This operation creates a new word from each of the input words joined together.

Normally only two inputs can be supplied to this operation. However, when enclosed in parentheses, the operation will be applied to any number of inputs.

```
? (word "anti "dis "establish "ment)
ANTIDISESTABLISHMENT
```

Word?   Word? *object*
                                             WordP *object*

Output TRUE if an object is a word.

```
? word? 100
TRUE

? word? "grape
TRUE

? word? [peach grape]
FALSE
```

Word? will output TRUE whenever its input is a word. Other types of inputs will output a FALSE. Remember that numbers are considered to be words.

WordP is the old way of spelling **Word?**. We include it for compatibility. The "P" stands for the word "predicate" which indicates that a TRUE or FALSE will be output.

Make turtle movements wrap at the edge of the
screen.

```
? wrap
```

Normally when the turtle hits the edge of the screen,
an error is generated. This command puts the turtle
into a special mode: when the turtle is moved off the
screen, it will wrap around to the other side. The
turtle will never leave the screen.

```
? wrap
? clearscreen
? left 30
? forward 10000
? position
[120.0 -134.3511]
```

Output the current X position of the turtle.

```
? home
? print xpos
0
```

This is the position of the turtle horizontally from the
home position. The positive direction is to the right;
negative is to the left.

If the screen is in the **Window** mode, the position
returned might be greater than can be displayed on
the screen.

Output the current Y position of the turtle.

```
? home
? print ypos
0
```

This is the position of the turtle vertically from the home position. The positive direction is upward; negative is downward.

If the screen is in the **Window** mode, the position returned might be greater than can be displayed on the screen.

Error Messages

Logo error messages are meant to help you quickly locate and understand the problems in your program. When an error occurs take a minute to carefully examine the error message.

This message will contain a string describing what error happened. If the error occurred in one of your procedures, Logo will also tell you the procedure name and print the erroneous line.

This section lists error messages by category. The categories are:

Undefined Objects

Procedure Definition

Procedure Inputs

Procedure Outputs

File Related

Arithmetic

Other

In the text below, error messages are printed in **bold letters**. The non-bold portions of the message are dependent on the actual error.

Note: Some types of errors will throw you completely out of Amiga Logo back to the Workbench screen. If this occurs, it is often possible to return to Amiga Logo simply by pressing the LEFT-AMIGA-M key sequence, or by clicking on the screen depth arrangers located in the upper right portion of your workbench screen (make certain that you are clicking an arranger for the screen, not a window).

Also, there is a class of errors, called *Internal Errors*, from which Logo cannot recover. If an internal error occurs, you will need to start Logo over. If you experience a particular type of internal error happening often, please report it to Commodore-Amiga.

Undefined Objects

These error messages will occur if you refer to a procedure, name, label, or catch before it has been defined (or in some cases after it has been erased).

If you are confused about the error, try using **PONs** or **POTs** to verify that the name you are using is defined and spelled as you thought.

I don't know how to something

You are trying to use a procedure that does not exist. A procedure name must be defined before you attempt to execute it.

You will also get this error if you misspell a primitive name. For example:

SETRGB2 [0 10 0]

I don't know how to SETRGB2

Something **is undefined**

You are trying to use a variable that does not exist. A variable must be defined before you attempt to access its value.

Cannot find label label

Logo cannot find the label you referred to in a **Go** command. Remember that you must put the label in the same procedure as the **Go** instruction; you can not use **Go** to jump from one procedure to another.

Cannot find catch for something

Logo cannot find the **Catch** name which corresponds to the **Throw** name you specified. You may have put the **Catch** in the wrong place; perhaps you did not execute the **Catch** prior to the **Throw**.

Procedure Definition

These error messages are related in some way to the definition of procedures.

Something is already defined

You are trying to create a procedure with a name that already exists. The name may not necessarily belong to another procedure; it may be a variable name. You will either need to name your procedure something else, or erase the existing name and try again.

Procedure is a primitive

You are trying to create a procedure with a name that is used by a Logo primitive. You will need to name your procedure something else (primitives cannot be erased).

Something can only be used in a procedure

Some commands can only be used inside a procedure: **Output, Stop, End, Go, Label, Local**. You attempted to use one of these commands outside of a procedure.

Procedure Inputs

These errors deal with wrong or missing procedure inputs.

Not enough inputs to procedure

You have not provided enough inputs to the specified procedure or primitive.

Procedure **does not like** object **as input**

The wrong type of object was input to the primitive.

Object **is not true or false**

Logo expected a true or false, but got another value.

Object **is not a word**

In this case you are trying to use something as a word that is not a word. Check your typing.

Too few items in object

There are not enough elements in an object to satisfy a request. For example, this could happen if you used **Item** on a list that was too short.

Procedure Outputs

These messages deal with procedure outputs.

I don't know what to do with something

A procedure output a value that was not input to anything.

Procedure **did not output**

Logo was expecting the procedure to output a value to be used as input by another procedure, but nothing was output.

File filename **was not found**

You attempted to load or erase a file that does not exist in the current directory.

Have you specified the correct device name and directories? If file is not in the current directory, you must use a proper path name. You may want to use **Dir** or **Catalog** to help locate the file name.

File filename **already exists**

You asked Logo to save a file that already exists. To correct this, you may either erase the file and resave it, or save it with a different name.

File filename **is locked**

You attempted to erase a file that was locked. This means the file is being accessed by another program, and you are not allowed to change it.

File filename **is wrong type**

The file you specified contains data which does not look like a Logo file. Check your file name.

Arithmetic

Cannot divide by zero

You attempted to divide a number by zero. This is not allowed; dividing by zero is undefined.

Number is out of range

You are dealing with a number that is too large for Logo, or is invalid for a particular primitive. Check your arithmetic.

These are the remaining error messages.

Cannot something from editor

This error occurs when you put editing commands within the text you are editing. If you do this and exit the editor, Logo attempts to execute the editing command, but to do so would interfere (and possibly erase) the current contents of the editor.

For example, executing an **Edit** command within the editor will cause this message.

Parentheses enclose too many expressions

If you get this message, you tried to put parentheses around more than just an expression. Parentheses are only used to group related expressions.

Unexpected ')'

Logo found a ')' without a matching '('. Check your typing.

Turtle out of bounds

This message indicates that you tried to move the turtle beyond the edge of the display while in the **Fence** mode. If you want to move the turtle off-screen you must use the **Window** mode. If you want the screen to wrap around, use the **Wrap** mode.

Remember that the edge of your screen is also determined by the screen resolution. Running at a lower resolution will reduce the working boundaries for the turtle.

Out of space

This message happens when you have used up nearly all available Amiga memory. To continue, you

will need to free some space by erasing procedures, names, or properties.

If you are executing other programs on your Amiga, they will be using up memory as well. You may want to try running Logo alone.

Also, the more colors or higher screen resolution you use, the more memory required. Try reducing the number of colors or resolution.

Cannot use printer

The printer is not available for use. Do you have it properly installed? Perhaps it's being used by some other Amiga task.

Printer did not work

For some reason the printer could not print what you requested. Check that the printer is on-line and has paper. If you were thrown back to the workbench screen, press LEFT-AMIGA-M to return to the Logo screen.

SPEAK device not available (Workbench 1.3)

This message occurs when you use the **Say** command. This command requires you to use a Workbench 1.3 system which has the SPEAK device present. If the SPEAK device is not mounted (not present in the system) this error will occur. SPEAK is normally mounted as part of the Amiga startup-sequence file. If you removed the line from this file that said "mount speak:", this may be the problem.

Index